Polygon is working with the community to find security vulnerabilities to keep the Polygon **Projects**, **Developers** and **Users** safe.

The bounty program is to ensure everyone has an opportunity to find bugs in the Polygon system, which will in turn help the team ensure the security of Polygon. Payouts will go up to $5,000 for critical issues.

Getting started

If you are new to blockchains and/or to Polygon, see Polygon Architecture.

Explore the code on GitHub. There are 3 main repositories for you to study:

- Heimdall
- Bor
- Contracts

Setting up

Set up a test network locally. See Running a node on the local environment

The Polygon CLI repository is an easy way to setup and manage the entire Polygon stack, including **Heimdall**, **Bor**, and the **Staking smart contracts** on a local environment. This would help in simulating tests and attacks locally.

If you want to run a full node on the Polygon Mainnet or Mumbai Testnet, you can follow the Run a Validator Node guide.

Obtaining tokens for testing

To get tokens, you can access the Polygon faucet and choose the Goerli network to get some tokens.

Questions?

Check out the forum and join the discussion on Discord.

- Forum
- Discord

You are encouraged to explore the following opportunities in our Bug Bounty Programs: * Immunefi POS: this program focuses on all Smart Contract and Blockchain assets related to Polygon PoS * Immunefi zkEVM: this program focuses on all Smart Contract and Blockchain assets related to Polygon zkEVM * HackerOne: this program focuses on all web applications and other assets under Polygon Labs

id: wiki-maintainers title: Wiki Maintainers sidebar_label: Wiki Maintainers description: A list of Polygon Wiki maintainers keywords: - docs - matic - polygon - wiki - docs - maintainers - contributors image: https://wiki.polygon.technology/img/polygon-logo.png slug: community-maintainers

The following is a list of regular contributors to the Polygon Wiki that include their contact information.

If one of these people ever contacts you directly, first check the contact information to see that it matches the origin of the message, and then feel free to get in touch with someone else on this list to verify the legitimacy of the original inquiry.

:::warning Employee impersonation is a common type of scam

Always validate the person you are in contact with and see if their contact information matches the origin of the message. It is also a good idea to get in touch with someone else on this list to verify the legitimacy of the original inquiry.

:::

Ashutosh Tripathi

Github: ashucoder9

Danny Salman

Github: dannys03

Vinícius Gomes

Github: cerberushades

id: contributor-guidelines title: How to Contribute sidebar_label: How to Contribute description: Prepare for your upcoming contribution keywords: - docs - matic - polygon - contribute - contributor - contributing image: https://wiki.polygon.technology/img/polygon-logo.png slug: orientation

:::tip Open a pull request

Feel free to raise an issue on the Polygon Wiki repository.

:::

Identify an area to Contribute to

There are several ways to identify an area where you can contribute to the Wiki:

- The easiest is to reach out to one of the Wiki maintainers saying "I want to help contribute to the Polygon Wiki". They'll work with you to find an area for you to contribute.
- If you have a specific contribution in mind but are unsure about it, confirm whether the contribution is appropriate by contacting one of the Wiki maintainers directly.
- If you do not have a specific contribution in mind, you can also browse the issues labelled as `help wanted` on the Polygon GitHub repos.
- Issues that additionally have the `good first issue` label are considered ideal for first-timers.

Add to the Polygon documentation

- If you need to add or change anything in Polygon Wiki, please raise a PR against the `master` branch (kindly check the sample PR).
- The documentation team will review the PR or reach out accordingly.
- Repository: https://github.com/maticnetwork/matic-docs

:::tip If you want to run our Wiki locally on your machine, check the section running the Wiki locally. If you are adding a new document, it is recommended to just have a basic summary/introduction and a link to your Github or documentation for more details. :::

Git rules

We use `gitchangelog` for all of our repos for change logs. For that, we need to comply with the following convention for commit message. There will be no merge if you are not following this convention.

Commit message convention

The following are suggestions to what might be useful to think about adding in your commit messages. You might want to separate roughly your commits into big sections:

- by intent (for example: new, fix, change ...)
- by object (for example: doc, packaging, code ...)
- by audience (for example: dev, tester, users ...)

Additionally, you could want to tag some commits:

- As "minor" commits that shouldn't get output to your changelog (cosmetic changes, small typo in comments...).
- As "refactor" if you don't really have any significative feature changes. Thus this should not also be part of the changelog displayed to final user for instance, but might be of some interest if you have a developer changelog.
- You could tag also with "api" to mark API changes or if it's a new API or similar.

Try writing your commit message by targeting user functionality as often as you can.

:::note Example

This is a standard git log `--oneline` to show how this information could be stored:

```
* 5a39f73 fix: encoding issues with non-ascii chars.
* a60d77a new: pkg: added ``.travis.yml`` for automated tests.
* 57129ba new: much greater performance on big repository by issuing only one shell command for all the commits. (fixes #7)
* 6b4b267 chg: dev: refactored out the formatting characters from GIT.
* 197b069 new: dev: reverse ``natural`` order to get reverse chronological order by default. !refactor
* 6b891bc new: add utf-8 encoding declaration !minor
```

:::

For more info please refer to What Are Some Good Ways to Manage a Changelog Using Git?.

For more details, see https://chris.beams.io/posts/git-commit/.

id: translate title: Help With Translations sidebar_label: Help With Translations description: Learn how to help translate the Polygon Wiki. keywords: - docs - matic - polygon - translate - translation - contribute - language image: https://wiki.polygon.technology/img/polygon-logo.png slug: translate

:::caution State of translations and localization efforts

We are committed to making the Polygon documentation accessible to a global audience and have added support for around fifteen languages, with more to come. However, the translations are still a work in progress, and we are working on automating the process to keep them up-to-date. We are also looking for native speakers in the community to help retranslate some sections and translate new content for the first time. Additionally, we are planning to support more languages in the future.

:::

:::info Help with translations

We are also currently exploring options for translation management and may reactivate our use of Crowdin shortly for community contributions. Thank you for your patience and support as we continue to improve the localization of our documentation.

Please see the **Docusaurus guide** in the meantime.

:::

id: tutorial-template title: General Tutorial Template sidebar_label: Tutorial Template description: Follow the tutorial template when writing a technical tutorial. keywords: - docs - matic - polygon - documentation - tutorial - contribute - template image: https://wiki.polygon.technology/img/polygon-logo.png slug: tutorial-template

This template should be used when contributing a tutorial to the Polygon Wiki. You may choose to contribute a tutorial on a topic of your choosing.

General guidelines

- The tutorial's scope should be clear from the title.
- The tutorial should be able to accurately describe the features and functionalities of product(s) or service(s).
- Try to keep the tutorial swift and concise, but expand on key concepts where appropriate. Give background information and further context when possible.
- For configuration and implementation steps, be specific.
- Please try your best to add supporting imagery, icons, or screenshots that complement the written content.

  > *The documentation team would also be happy to work with you on creating diagrams.*

- Remember the audience you are writing for. If the material has a certain difficulty level it should be mentioned in the tutorial.

  > *If there are steps a user should take before running through a tutorial, please mention them.*

- The documentation would be happy to jointly work on creating a tutorial.
- Remember to consider the **Style guide**.

:::caution Updating current tutorials

If you notice that the current tutorials on the Polygon Wiki do not follow this template, it is because the documentation team decided to implement a standard, so the tutorial flow is consistent across all tutorials. The team is working on updating these tutorials to resemble this template. If you are interested, you can also update an existing tutorial to restructure it.

:::

Tutorial sections

Overview

Explain the product(s) or service(s) being discussed in the tutorial. Give background information for the purpose of the tutorial and what the tutorial aims to present. The tutorial should always be based on using a Polygon product.

What you'll learn

Summarize what the user will learn throughout the tutorial.

:::note Example

You'll explore how to use the Truffle Suite to build dApps Polygon.

:::

Learning outcomes

Outline the learning outcomes.

:::note Example

1. You will learn about Fauna.
2. You will learn how you can use ReactJS for the UI of your dApp.
3. You will learn how to safeguard dApp data.

:::

Mention prerequisites and what the user should already be familiar with. Link the necessary documentation for areas that the user should already be knowledgeable about.

:::note Example

Before starting this tutorial, you should understand the basics of EVM-based dApp development. See "these docs" for more information.

:::

What you'll do

Outline the steps in the tutorial and the tools that will be used.

:::note Example

You will use Solidity to create a smart contract in a ChainIDE environment.

1. Setting up a wallet
2. Write an ERC-721 smart contract
3. Compile an ERC-721 Smart Contract

4. Deploy an ERC-721 Smart Contract
5. Create a Flattened File using Flattener Library
6. Verify a Smart Contract
7. NFT Minting

:::

The tutorial itself

In general, the tutorial can be presented in the best categorization that the writer sees fit; this should be reflected in the What you'll do section. However, the tutorial sections should touch on these three main categories:

> *Ensure that you consider keywords and keep SEO in mind when coming up with sections.*

Build your application

The main tutorial content. This can include sections like "setup", "configuration", and "implementation" to name a few.

Run or Deploy your application

Explain how the user should run or deploy their application.

Test your application

This could be writing tests for a smart contract, smart contract verification, etc.

Next steps

Conclude the tutorial and reflect on the learning outcomes. Outline next steps that the user can take.

:::note Example

Congratulations on deploying your smart contract. You now know how to use ChainIDE to create and deploy smart contracts. Consider trying out "this tutorial".

:::

id: writing-style title: General Writing Guidelines sidebar_label: Writing Guidelines description: Follow the following guidelines when writing. keywords: - docs - matic - polygon - documentation - writing - contribute - style image: https://wiki.polygon.technology/img/polygon-logo.png slug: writing-style

This guideline focuses on best practices for writing technical documentation and on the style conventions to use when developing documentation for the Polygon Wiki. The goal of this guide is to help contributors write content that is clear, concise, and consistent. The Polygon team treats the Polygon Wiki as an official Docs product.

Primary guidelines

We believe one of the things that makes Polygon special is its coherent design and we seek to retain this defining characteristic. The Polygon team treats the Polygon Wiki as an official Docs product. From the outset we defined some guidelines to ensure new contributions only ever enhance the overall project:

- **Quality**: Code in the Polygon project should meet the style guidelines, with sufficient test-cases, descriptive commit messages, evidence that the contribution does not break any compatibility commitments or cause adverse feature interactions, and evidence of high-quality peer-review.
- **Size**: The Polygon project's culture is one of small pull-requests, regularly submitted. The larger a pull-request, the more likely it is that you will be asked to resubmit as a series of self-contained and individually reviewable smaller PRs.
- **Maintainability**: If the feature will require ongoing maintenance (e.g. support for a particular brand of database), we may ask you to accept responsibility for maintaining this feature.

The Style guide takes motivation from the following style manuals:

> *If you are unable to find the answer to a style, voice, or terminology question in this guide, please consult these resources.*

- Google's Style Guide
- The Oxford Style Manual
- The Microsoft Manual of Style

Static-site generator

The Wiki is built using Docusaurus, a static-site generator for building documentation sites in markdown. The Wiki follows the following metadata template for its markdown files and should be adapted for each new document:

```
---
id: wiki-maintainers
title: Wiki Maintainers
sidebar_label: Maintainers
description: A list of Polygon Wiki maintainers
keywords:
  - docs
  - matic
  - polygon
  - wiki
  - docs
  - maintainers
  - contributors
```

```
image: https://matic.network/banners/matic-network-16x9.png
slug: community-maintainers
---
```

There are some important aspects to consider when writing the metadata for a markdown file:

- We ask contributors to use a **unique id**; avoid **only** using generic words or sentences like "Introduction" or "Overview".
- The **title** is the sentence used at the beginning of the article, "General Writing Guidelines" for this article. So, it is not necessary to add an H1/H2 header to introduce each article. Instead, use this **title** from the metadata.
- The **description** can not be too lengthy, since it is used on the index tiles which has a limitation for the number of characters. For example, the

  description "Blockchain is an immutable ledger for recording transactions" for the *basics-blockchain.md* appears on an index tile as such: img

  The **description** then should have **up to 60 characters**, considering spaces between characters.
- Keywords are important to increase SEO and describe the article. Try to aim for at least five keywords.

:::note

Please see the [official metadata documentation]() for more details.

:::

Share the experience with the reader

- First Person: Do not use "I" or "me". Use the first person point of view sparingly and with intention. When overused, the first person narrative can overwhelm the sense of a shared experience and obscure the reader's journey.
- Second Person: In most cases, address the reader directly. For tutorials, use either first person plural—we, us, our, ours—or second person point of view. Because tutorials provide a more guided approach to a topic, using the first person plural is a more natural and commonly-accepted practice than in other types of documentation.
- Third Person: Do not use "we" to refer to Polygon or Polygon Technology.
- Active Voice: Use present tense whenever possible. There are situations where a passive voice is appropriate; revert to passive voice when the agent needs to be the focus.
- Keep the human presence in mind: having a dynamic tone when describing technical concepts really helps a reader connect with the material as opposed to describing software (or code) only for what it does.
- Pronouns: use gender-neutral pronouns, like "they" whenever possible. Generally, you can change any noun from singular to plural to have subject-verb-pronoun agreement and avoid the use of gender-specific pronouns like "he", "him", "his" or "she", "her", "hers".
  - Be wary of impersonal and potentially ambiguous pronouns. If you use any of the following impersonal pronouns, be sure you answer "of what?", "of which?", or "as what?" in the sentence.
    - all, another, any
    - each, either
    - few, many, neither, none,
    - one, other
    - same, several, some, such
    - that, them, these, those

Being swift and concise

- Documentation is strong and meaningful when the necessary words and right phrases are used.
  - Use common, well-known words whenever possible.
  - Avoid flowery language and excessive literary phrases.
  - Avoid jargon, colloquialisms, and idiomatic phrases.
  - Avoid adverbs and subjective statements. For example, don't use words and phrases that include easily, rapidly, simply, quickly. If need be, it is also better to underexaggerate than to overexaggerate.
  - Don't use phrases that introduce ambiguity. For example, instead of "When this release is live..." use "After this release is live...".
  - Pay extra attention to with word choice. Choosing to use "since" (implying a period of time) instead of "because" (implying cause and result) or using "once" (single occurrence) instead of "after" (each time).
  - Avoid exclamation marks.
- Avoid adding unnecessary words or phrases. Some examples:
  - Rather than saying "and then", just use "then".
  - Rather than saying "In order to", just use "to".
  - Rather than saying "as well as", just use "and".
  - Rather than saying "via", use an appropriate English substitute such as "using", "through", or "by".
- Use a conversational tone that is not too formal, but should still be professional.
- Clarity: give life to the word or phrase where possible. For example:
  - Rather than saying "e.g.", use "for example".
  - Rather than saying "i.e.", use "that is" or rewrite the sentence to make the meaning clear without needing extra qualification.
  - Rather than saying "etc.", use "and so on" or revise the content to make the term unnecessary. Instead of "etc." to end a list of examples, focus on an example or two and use "such as" or "like".
  - Instead of "caveat", use an appropriate English substitute such as "notice", "caution", or "warning".
  - Contractions give documentation a more natural conversational tone—at least for English speakers. Be conscious of when and why you use contractions.

Structure

Documents should be organized in sections. Each section should be responsible for presenting a theme or topic. Within each section, one or multiple paragraphs will exist. Each paragraph should convey only one thought. Try to avoid repeating the same thoughts in different sections, and split paragraphs that have multiple

points of discussion. The reader should understand what a paragraph is about from its first sentence.

Product documentation

If you are writing about a specific product, ensure the document resembles that product. Previously, the Polygon documentation was generalized, based around Polygon PoS. Now that there are multiple Polygon-based products, contributors need be wary about their additions.

For instance, "Deploying a smart contract on Polygon using ####" is ambiguous. If this tutorial was referring to Polygon PoS, it should be clear, as in, "Deploying a smart contract on Polygon PoS using ####". Using the same example with a Polygon Rollup, like Polygon Hermez, "Deploying a smart contract on Polygon Hermez using ####".

Ensure that the product documentation, whether a general guide or tutorial, is added to the right product documentation Hub. For most documents, their reference should exist under one of the general Hubs (e.g. "Develop" or "Validate"), but the actual document will live under its product documentation. You will need reference the document in the Hub by adding it to `sidebars.js`. However, the actual document itself will exist in its respective product documentation Hub, and it will redirect the user once they click on it. The same guideline applies to most documents. Their reference should exist under one of the general Hubs, but the actual document will live under its product documentation.

Most of the API-based documentation on the Polygon Wiki are in the form of reference documentation, with the exception of the APIs mentioned in tutorials. For instance, the API documentation on Matic.js provides information about the structure, parameters, and return values for each function or method in the API.

API documentation

Consider the following when documenting an API:

- A solid introduction that provides a starting point.
- A clear description of the call or request. Describe what the endpoint does.
- A full parameter list:
  - Parameter types
  - Syntax expressions with placeholders showing available parameters
  - Special formatting
- Code examples for multiple languages.
- A sample call with the expected output.
- Error Codes. Edge cases.
- Instructions on how to acquire API keys, if needed.
- Noting common FAQs or scenarios is always useful.
- Links to additional resources such as social media posts, blogs, or video content.

---

id: delegate title: How to Delegate description: Learn how to become a delegator on the Polygon Network. keywords: - docs - matic - polygon - how to delegate - validator - stake image: https://wiki.polygon.technology/img/polygon-logo.png slug: delegate

import useBaseUrl from '@docusaurus/useBaseUrl';

How to Delegate

This is a step-by-step guide to help you become a delegator on the Polygon Network.

The only prerequisite is to have your MATIC tokens and ETH on the Ethereum mainnet address.

Access the dashboard

1. In your wallet (e.g. MetaMask), choose the Ethereum mainnet.

2. Log in to Polygon Staking.
3. Once you log in, you will see some overall statistics along with the list of validators.

img

:::note

If you are a validator, use a different non-validating address to log in as delegator.

:::

Delegate to a Validator

1. Click **Become a Delegator** or scroll down to a specific validator and click **Delegate**.

img

2. Provide the amount of MATIC to delegate.

3. Approve the delegation transaction and click **Delegate**.

After the delegation transaction completes, you will see the **Delegation Completed** message.

View your delegations

To view your delegations, click My Account.

img

Withdraw rewards

1. Click My Account.

2. Under your delegated validator, click **Withdraw Reward**.

This will withdraw the MATIC token rewards to your Ethereum address.

Restake rewards

1. Click My Account.

2. Under your delegated validator, click **Restake Reward**.

This will restake the MATIC token rewards to the validator and increase your delegation stake.

Unbond from a Validator

1. Click My Account.

2. Under your delegated validator, click **Unbond**.

This will withdraw your rewards from the validator and your entire stake from the validator.

Your withdrawn rewards will show up immediately on your Ethereum account.

Your withdrawn stake funds will be locked for 80 checkpoints.

:::note

The fund locking for the unbonding period is in place to ensure there is no malicious behaviour on the network.

:::

Move stake from one node to another node

Moving stake from one node to another node is a single transaction. There are no delays or unbonding periods during this event.

1. Log in to the My Account on the Staking dashboard.
2. Click **Move Stake** under your delegated validator.
3. Select an external validator and click **Stake here**.

4. Provide the stake amount and click **Move Stake**.

This will move the stake. The dashboard will update after 12 block confirmations.

:::info

Moving stake is allowed between any nodes. The only exception is moving stake from one Foundation node to another Foundation node which is not allowed.

:::

id: delegator-faq title: Delegator FAQ sidebar_label: Delegator FAQ description: FAQs related to Delegation on Polygon network keywords: - docs - polygon - how to delegate - validator - stake - faq - delegator image: https://wiki.polygon.technology/img/polygon-logo.png

import useBaseUrl from '@docusaurus/useBaseUrl';

What is the Staking Dashboard URL?

The staking dashboard URL is https://staking.polygon.technology/.

What is the minimum stake amount?

There is no minimum stake amount to delegate. However, you can always start with 1 MATIC token.

How many rewards will I get if I delegate?

Please use the Staking Rewards Calculator to determine your estimates.

Why does my transaction take so long?

All staking transactions of Polygon happen on Ethereum for security reasons.

The time taken to complete a transaction depends on the gas fees that you have allowed and also the network congestion of Ethereum mainnet at that point in time. You can always use the “Speed Up” option to increase the gas fees so that your transaction can be completed soon.

Which wallets are currently supported?

Currently, only the MetaMask extension on the desktop browser and Coinbase Wallet are supported. Additionally you can use WalletConnect and Walletlink from supported mobile wallets to interact with the Staking UI dashboard on desktop / laptop. We will be gradually adding support for other wallets soon.

Are hardware wallets supported?

Yes, hardware wallets are supported. You can use the "Connect Hardware Wallet" option on MetaMask and connect your Hardware wallet and then continue the delegation process.

Why can’t I stake directly from Binance?

Staking through Binance is not yet supported. There will be an announcement if and when Binance starts supporting it.

I have completed my delegation, where can I check details?

Once you have completed your delegation, wait for 12 block confirmations on Ethereum (approx. 3-5 minutes), then on the Dashboard, you can click on **My Account**.

Where can I check my rewards?

On the Dashboard, you can click on the **My Account** option on the left-hand side.

Do I need ETH to pay for Gas fees?

Yes. You should provision for ~0.05-0.1 ETH to be safe.

Do I need to deposit MATIC tokens to the Polygon Mainnet network for staking?

No. All your funds need to be on the Main Ethereum Network.

When I try to do the transaction my Confirm button is disabled, why so?

Please check if you have enough ETH for the gas fees.

When does reward get distributed?

The rewards are distributed whenever a checkpoint is submitted.

Currently, 20188 MATIC tokens are distributed proportionately on each successful checkpoint submission to each delegator based on their stake relative to the overall staking pool of all validators and delegators. Also, the percentage for the reward distributed to each delegator will vary with each checkpoint depending on the relative stake of the delegator, validator and the overall stake.

(Note that there is a 10% proposer bonus that accrues to the validator who submits the checkpoint, but over time, the effect of the extra bonus is nullified over multiple checkpoints by different validators.)

The checkpoint submission is done by one of the validators approximately every 34 minutes. This time is approximate and may vary based on validator consensus on the Polygon Heimdall layer. This may also vary based on Ethereum Network. Higher congestion in the network may result in delayed checkpoints.

You can track checkpoints on the staking contract here

Why does reward keep getting decreased every checkpoint?

Actual rewards earned will depend on the actual total locked supply in the network at each checkpoint. This is expected to vary significantly as more MATIC tokens get locked in the staking contracts.

Rewards will be higher, to begin with, and will keep decreasing as the locked supply % goes up. This change in locked supply is captured at every checkpoint, and rewards are calculated based on this.

How can I claim my rewards?

You can claim your rewards instantly by clicking on the **Withdraw Reward** button. This will transfer the rewards accumulated to your delegated account on Metamask.

What is the Unbonding period?

The unbonding period on Polygon is approximately 9 days now. It was 19 days previously. This period applies to the originally delegated amount and re-delegated amounts - it does not apply to any rewards that were not re-delegated.

Will I keep receiving rewards after I unbond?

No. Once you unbond, you will stop receiving rewards.

How many transactions does the delegation require?

Delegation requires 2 transactions, one after the other. One to **Approve** the request and another to **Deposit**.

What does Redelegate Rewards mean?

Redelegating your rewards simply means that you want to increase your stake by restaking the rewards you have accumulated.

Can I stake to any validator?

Yes. All validators are Polygon Foundation nodes currently.

We are doing a phased rollout of the Polygon Mainnet. Later on, external validators will be onboarded gradually. Please see https://blog.matic.network/mainnet-is-going-live-announcing-the-launch-sequence/ for more details.

Which browser is compatible with Staking Dashboard?

Chrome, Firefox, and Brave

My MetaMask is stuck at confirming after login, what do I do? Or nothing happens when I try to login?

Check for the following:

- If you're using Brave, please turn off the option for **Use Crypto Wallets** in the settings panel.
- Check if you are logged into Metamask
- Check if you are logged into MetaMask with Trezor/Ledger. You need to additionally turn on permission to call contracts on your Ledger device, if not enabled already.
- Check your system timestamp. If the system time is not correct, you will need to correct it.

How do I send funds from Binance or other exchanges to Polygon wallet?

Technically, the Polygon Wallet Suite/Staking interface is just a web application. Currently it supports the following wallets - Metamask, WalletConnect, and WalletLink.

First, you must withdraw your funds from Binance or any other exchange to your Ethereum address on Metamask. If you don't know how to use Metamask, google it a bit. There are plenty of videos and blogs to get started with it.

When can I become a validator and how many tokens do I for that?

A user can earn a validator spot only by if the below conditions come into play:

1. When a validator decides to unstake from the network, or
2. Wait for the auction mechanism and replace the inactive validator.

The minimum stake depends on the auction process where one user outbids another.

If I have earned rewards while delegating, and if I add additional funds to the same validator node, what happens?

If you have not re-delegated your rewards before delegating additional funds to the same validator node, your rewards will be withdrawn automatically.

In case you dont want that to happen, re-delegate your rewards before delegating additional funds.

I have delegated my tokens via MetaMask on the Staking dashboard. Do I need to keep my system or device on?

No. Once your Delegation transactions are confirmed, and you can see your tokens reflected in the **Total Stake** and **New Reward** sections, then you are done. There is no need to keep your system or device on.

I have unbonded, how long will it take to Unbond?

The unbonding period is currently set to 82 checkpoints. This is approximately 9 days. Every checkpoint takes approximately 34 minutes. However, some checkpoints could be delayed upto ~1 hour due to congestion on Ethereum.

I have unbonded, and I now see the Claim Stake button, but it is disabled, why is that?

The Claim Stake button will only be enabled when your unbonding period is complete. The unbonding period is currently set at 82 checkpoints.

Do I know when will the Claim Stake button be enabled?

Yes, under the Claim Stake button you would see a note on how many checkpoints are pending before the Claim Stake button would be enabled. Every checkpoint takes approximately 30 minutes. However, some checkpoints could be delayed upto ~1 hour due to congestion on Ethereum.

How do I switch my delegation from Foundation Nodes to External nodes?

You can switch your Delegation using the **Move Stake** option on the Staking UI. This will switch your Delegation from the Foundation node to any other external node of your choice.

You will see a list of other validators:

Will there be any ubonding period when I switch Delegation from Foundation nodes to external nodes?

There will be no Unbonding period when you switch Delegation from foundation nodes to external nodes. It will be a direct switch without any delays. However, if you are unbonding from a Foundation Node or an External node there will be an Unbonding period for that.

Are they any specifics to choose an external node during switch delegation?

No. You can choose any node of your choice.

What happens to my rewards that are accumalated if I switch delegation from Foundation to External node?

If you haven't already claimed your rewards before switching delegation, then upon successful switch of your delegation from Foundation to External the Rewards that were accumalated till then will be transferred back to your account.

Will delegation on the External Nodes work the same as Foundation Nodes?

Yes, it will work the same as Foundation nodes.

Will I still get rewards after delegating to an External Node?

Yes, rewards will be distributed the same as earlier with the Foundation nodes. Every successful submission of a checkpoint will yield in rewards. Rewards will be distributed and calculated at every checkpoint relative to the stake ratio, as currently implemented.

Will there be any unbonding period if I unbond from an External Node?

Yes, the unbonding period will stay the same as currently implemented. 82 Checkpoints.

Will there be any locking period after I switch my Delegation from Foundation to External node?

No. There won't be any locking period after you switch your delegation.

Can I partially switch my delegation from Foundation to External nodes?

Yes, you will have the option to partially move your stake from Foundation node to an external node. The remaining partial stake will remain on the Foundation node. You can then move that to another node of your choice or the same node.

Can I switch delegation from an external node to another external node?

No, the **Move Stake** option is only available on the Foundation Nodes. If you want to switch your delegation from an external node to another external node, you will have to unbond first and then delegate to another external node.

When will the Foundations node be turned off?

The foundation nodes will be turned off by the end of January, 2021.

Will there be any Foundation nodes in the future?

No, there won't be any Foundation nodes in the future.

How many transactions do I need to pay for Gas when I do a Move Stake?

The Move Stake is a single transaction only. All transactions would be on the Ethereum Blockchain so you would need to spend some ETH while doing the Move Stake transaction.

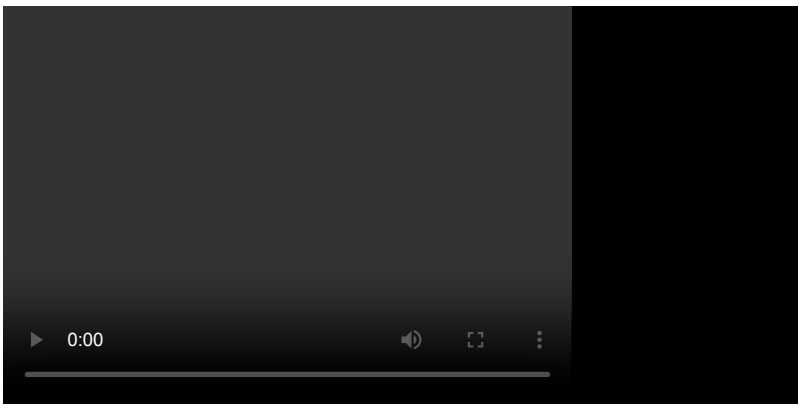
id: staking-faq title: Staking FAQ sidebar_label: Staking FAQ description: Build your next blockchain app on Polygon. keywords: - docs - matic image: https://wiki.polygon.technology/img/polygon-logo.png

import useBaseUrl from '@docusaurus/useBaseUrl';

How to stake tokens on Polygon?

For Staking you would need to have funds on the Ethereum Mainnet (more information here). Log into MetaMask on the Ethereum network using the Staking Dashboard. https://staking.polygon.technology/

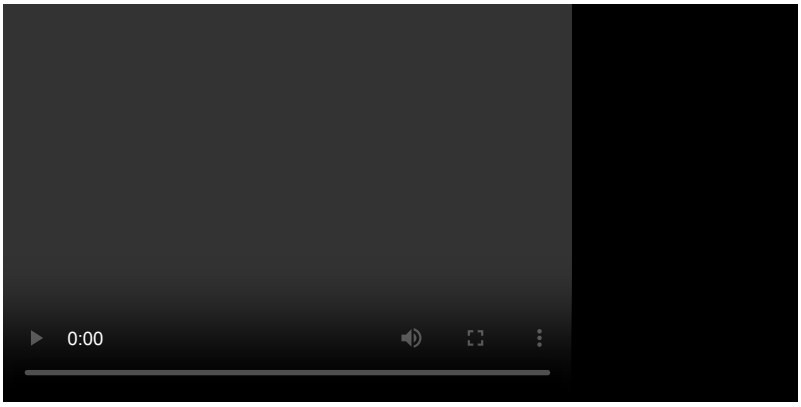
Please watch this video for a graphical illustration of how this works:



I've staked my Matic tokens. How can I stake more?

You can navigate to "Your Delegations", choose one of the stakes and click on "Stake More".

Please watch this video for a graphical illustration of how this works:



Why am I not able to stake?

Check if you have funds on the Main Ethereum Network, to delegate your tokens. All staking happens on the Ethereum Network only.

I am unable to view the Staking tab. How do I access Staking?

You just need to access **https://staking.polygon.technology/**, where you will see the following landing page:

<img src={useBaseUrl("img/staking_faq/staking-lp.png")} height="500px"/>

Alternatively, once logged into the Polygon Wallet Suite, you can click on **More Apps > Staking**. Users will be landed on the **Staking Overview** page. Reference for guide:

<img src={useBaseUrl("img/staking_faq/staking-app.png")} height="500px"/>

How do I know which Validator to select for better rewards?

It depends on your understanding and research on which validator you would want to stake on. You can find the list of validators here : https://staking.polygon.technology/validators

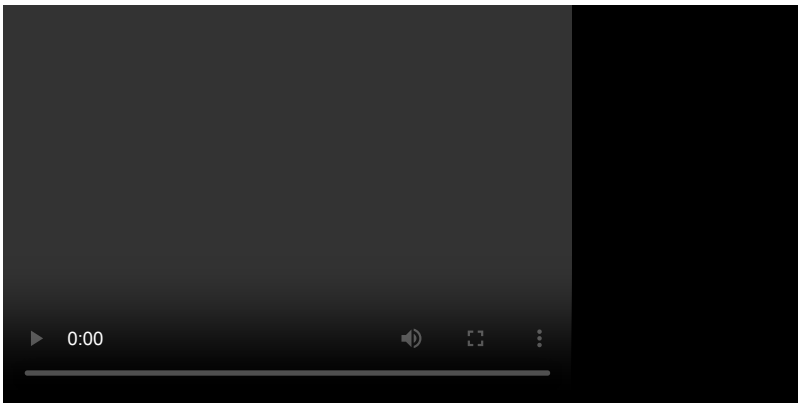
How to unbond?

To Unbond from a Validator, navigate to MyAccount, where you find "Your Delegations". There you will see an Unbond button for each of the validators. Click on the Unbond button for whichever validator that you want to unbond from.

`Step 1`
<img src={useBaseUrl("img/staking_faq/step1unbond.png")} height="400px"/>

`Step 2`
<img src={useBaseUrl("img/staking_faq/step2unbond.png")} height="500px"/>

Please watch the video for a graphical illustration of how this works:



What is the unbonding period?

The unbonding period on Polygon is 80 checkpoints. This is approximately ~3-4 days. Every checkpoint takes approximately 3 hours. However, some checkpoints could be delayed due to congestion on Ethereum. This period applies to the originally delegated amount and re-delegated amounts. It does not apply to any rewards that were not re-delegated.

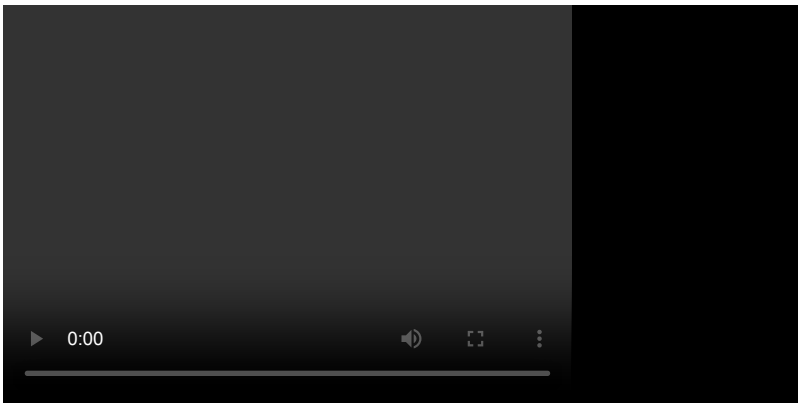
How to restake rewards?

Go to “My Account” to check "Your Delegations". Clicking on "Restake Reward" will ask you for confirmation from your MetaMask account. Once you confirm the transaction, only then the restake transaction would be complete.

`Step 1`
<img src={useBaseUrl("img/staking_faq/restake-rewards1.png")} height="300px"/>

`Step 2`
<img src={useBaseUrl("img/staking_faq/restake-rewards2.png")} height="415px"/>

Please watch the video for a graphical illustration of how this works:



I want to Restake Rewards but I am unable to.

You would need to have a minimum of **2 Matic** to restake rewards.

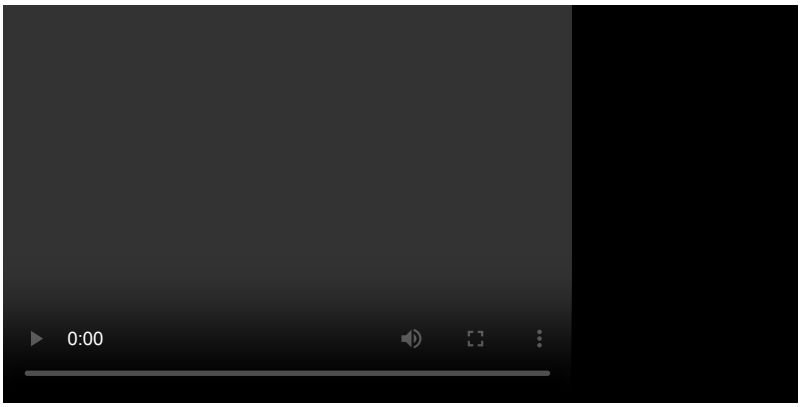
How to withdraw rewards?

You can claim your rewards by clicking on the “My Account”, all the delegators for a validator are displayed. Click on the “Withdraw Reward” button and the rewards will be transferred to your delegated account on Metamask.

`Step 1`
<img src={useBaseUrl("img/staking_faq/withdraw1.png")} height="300px"/>

`Step 2`
<img src={useBaseUrl("img/staking_faq/withdraw2.png")} height="380px"/>

Please watch the video for a graphical illustration of how this works:



I want to Withdraw Rewards but I am unable to.

You would need to have a minimum of **2 Matic** to withdraw rewards.

How to claim stake?

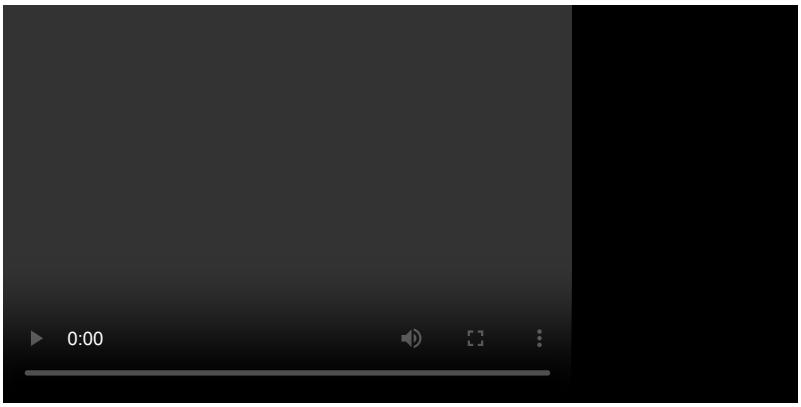
Once the **unbonding period is complete**, the Claim Stake button will be enabled and you can then claim your staked tokens. The tokens will be transferred to your account.

`Step 1`
<img src={useBaseUrl("img/staking_faq/claim-stake1.png")} height="400px"/>

`Step 2`
<img src={useBaseUrl("img/staking_faq/claim-stake2.png")} height="300px"/>

`Step 3`
<img src={useBaseUrl("img/staking_faq/claim-stake3.png")} height="400px"/>

Please watch the video for a graphical illustration of how this works:



Are hardware wallets supported?

Yes, hardware wallets are supported. You can use the "Connect Hardware Wallet" option on MetaMask and connect your Hardware wallet and then continue the delegation process.

Why can’t I stake directly from Binance?

Staking through Binance is not yet supported. There will be an announcement if and when Binance starts supporting it.

Do I need to deposit Matic tokens to the Polygon Mainnet network for staking?

No. All your funds need to be on the Main Ethereum Network.

When do rewards get distributed?

The rewards are distributed whenever a checkpoint is submitted.

Approximately 20188 Matic tokens are distributed proportionately on each successful checkpoint submission to each delegator based on their stake relative to the overall staking pool of all validators and delegators. Also, the percentage for the reward distributed to each delegator will vary with each checkpoint depending on the relative stake of the delegator, validator and the overall stake.

(Note that there is a 10% proposer bonus that accrues to the validator who submits the checkpoint, but over time, the effect of the extra bonus is nullified over multiple checkpoints by different validators.)

The checkpoint submission is done by one of the validators approximately every 34 minutes. This time may vary based on validator consensus on the Polygon Heimdall layer. This may also vary based on the Ethereum Network. Higher congestion in the network may result in delayed checkpoints.

You can track checkpoints on the staking contract here: https://etherscan.io/address/0x86e4dc95c7fbdbf52e33d563bbdb00823894c287

Why do rewards keep getting decreased at every checkpoint?

Rewards earned will depend on the actual total locked supply in the network at each checkpoint. This is expected to vary significantly as more MATIC tokens get locked in the staking contracts. Rewards will be higher, to begin with, and will keep decreasing as the locked supply percentage goes up. This change in locked supply is captured at every checkpoint, and rewards are calculated based on this.

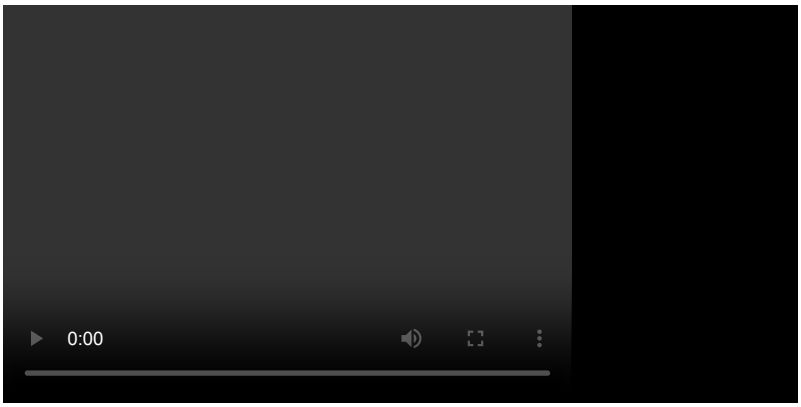
Will I keep receiving rewards after I unbond?

No. Once you unbond you stop receiving rewards.

Can I move the stake to another validator?

Yes, you just have to reach "Your Delegations", click on "Move Stake", and then choose your new validator.

Please watch the video for a graphical illustration of how this works:



Which browsers are compatible with the Staking Dashboard?

Chrome, Firefox, and Brave.

Nothing happens when I try to log in or my MetaMask is stuck at confirming after logging in. What do I do?

Check for the following: - If you're using Brave, please turn off the option for "Use Crypto Wallets" in the settings panel. - Check if you are logged into Metamask. - Check if you are logged into MetaMask with Trezor/Ledger. You need to additionally turn on permission to call contracts on your Ledger device, if not enabled already. - Check your system timestamp. If the system time is not correct, you will need to correct it.--- id: governance-pos title: Polygon PoS Consensus and Governance sidebar_label: PoS Governance description: Common governance terms and processes for the Polygon PoS blockchain. keywords: - docs - matic - polygon - DAO - governance slug: governance-pos image: https://wiki.polygon.technology/img/polygon-logo.png

This page includes common governance terms and processes for the Polygon PoS blockchain.

Heimdall

Upgrade Type	Softfork/Hardfork	Proposal	Consensus Options	Implementation Flow
Software upgrades	Both	PIP proposed via community forum (recommended)	Rough consensus	Rough consensus: 1. Development and testing on local dev machines 2. Multi-node testing on devnet 3. Rolling out the release on a few Mumbai nodes (testnet) 4. Rolling out the release on a few Mainnet nodes 5. Full roll-out on Mumbai (testnet) 6. Full roll-out on Mainnet
Consensus rules	Hardfork only	PIP proposed via community forum (recommended)	Rough consensus	Rough consensus: 1. Development and testing on local dev machines 2. Multi-node testing on devnet 3. Rolling out the release on a few Mumbai nodes (testnet) 4. Rolling out the release on a few Mainnet nodes

				5. Full roll-out on Mumbai (testnet) 6. Full roll-out on Mainnet
Governance module	Both	PIP proposed via community forum (recommended)	Governance module vote	Governance module: 1. Proposal submission: Validators can submit proposals with a deposit. Once the minimum deposit is reached, the proposal enters the voting period. Validators that deposited on proposals can recover their deposits once the proposal is rejected or accepted. 2. Vote: Validators can vote on proposals that reached MinDeposit 3. Once the voting parameters needed to pass the proposal are met, the change is simultaneously carried out across the network

Bor

Upgrade Type	Softfork/Hardfork	Proposal	Consensus Options	Implementation Flow
Software upgrades	Both	PIP proposed via community forum (recommended)	Rough consensus	Rough consensus: 1. Development and testing on local dev machines 2. Multi-node testing on devnet 3. Rolling out the release on a few Mumbai nodes (testnet) 4. Rolling out the release on a few Mainnet nodes 5. Full roll-out on Mumbai (testnet) 6. Full roll-out on Mainnet
Consensus rules	Hardfork only	Upgrades are made in line with the geth implementation	Rough consensus	Rough consensus: 1. Development and testing on local dev machines 2. Multi-node testing on devnet 3. Rolling out the release on a few Mumbai nodes (testnet) 4. Rolling out the release on a few Mainnet nodes 5. Full roll-out on Mumbai (testnet) 6. Full roll-out on Mainnet

Definitions

Hardfork

A hardfork happens when the node software changes, so the new version is no longer backward-compatible with earlier blocks. This is usually the result of a change in the consensus calculation, meaning that blocks validated using the new calculation will produce a different hash.

In the current style of change implementation, hardfork block numbers are broadcasted by the Polygon team after an initial staggered rollout to the larger nodes. A block number is selected before which all nodes in the network should have upgraded to the new version. Nodes running the old version will stop working (will be disconnected from the canonical chain after the hardfork block).

Should there be ⅓+1 staked MATIC in disagreement with the fork, two canonical chains will temporarily form until the end of the current span. After this, Bor will stop producing blocks, and the chain will halt until consensus is reached.

Softfork

This type of change is backward-compatible with the pre-fork blocks. This protocol change does not require nodes to upgrade before a deadline. Therefore multiple versions of the node software can be running at once and be able to validate transactions.

Rough Consensus

Defined as the 'dominant view' of a group as determined by the current consensus framework. Without a vote that can carry out a synchronous update across the network, the 'dominant view.' is defined by the node software used by each validator, weighted by its total stake.

In the case of the Polygon PoS chain, this is programmatically defined by ⅔+1 of total staked MATIC.

Governance module

The Polygon PoS consensus engine (Heimdall) has an inbuilt governance module that can synchronously carry consensus parameter changes across the network. Users can submit proposals to the module along with a deposit containing the proposed changes. Each validator tallies votes cast by validators. Once the defined voting parameters are met, each validator automatically updates itself with the proposal data.

The current voting parameters (denominated in staked MATIC):

- Quorum: 33.4%
- Threshold: 50%

- Veto: 33.4%

A list of the changeable parameters by the Heimdall governance module is available here.

Polygon Improvement Proposal (PIP)

The PIP process, as defined in PIP-1, outlined a preliminary approach for allowing the community to put forward-protocol upgrades that improve the network.

This takes inspiration from the PEP process, which has become a standard used for open source projects, including Ethereum and Bitcoin.

As a guiding rule, it is recommended that all changes originate from the Polygon community Forum and the PIP Github repository, in the form of a proposal written in line with PIP-1.

On the Polygon PoS chain, different classifications of PIPs have different implementation flows. Thus, it is suggested to consult with the Governance processes board regarding the implementation of different types of PIPs.

id: pips-community-guidelines title: Community Guidelines description: Polygon Community Guidelines. keywords: - docs - matic - polygon - ethereum image: https://wiki.polygon.technology/img/polygon-logo.png

Polygon Community Forum Guidelines

At Polygon, we are committed to creating a decentralized, open-source, and accessible-for-all web3 space and this starts with an open community filled with positivity, active engagement, open discussion of ideas, and mutual respect.

Community Culture

The Polygon Community Forum, and the broader community that uses the platform, are dedicated to developing, improving, and further decentralizing the Polygon ecosystem. Our community culture is one of openness, inclusiveness, and positivity.

Core Values

The rules and guidelines of the forum reflect the core values of the Polygon ecosystem.

Ecosystem First

We always prioritize the health of the ecosystem and community above particular interests. Any intentional disruption or act that endangers the well-being of the Polygon community will not be tolerated.

Openness

The forum is open, accessible, and transparent by its nature. Feel free to share your thoughts and ideas on anything and everything related to Polygon.

Growth

We are here to build and advance Polygon's scaling solutions, but we are also here to build ourselves and each other. Ultimately, if you find yourself wondering, "should I post this?" Ask yourself whether it aligns with the values and guidelines below.

Guidelines

Much like the blockchain itself, the Polygon community does not discriminate against participants based on their color, sexual orientation, gender, nationality, or background. Therefore, we will not tolerate any derogatory comments whatsoever, including those regarding users' race, gender, sexual orientation, ethnicity, or age.

This is not an exhaustive list, and you should keep in mind the following guidelines when posting anything on the forum.

When on the Polygon Community Forum, you should:

1. Be tolerant towards others' viewpoints. When you disagree, do it respectfully.
2. Communicate with courtesy and respect.
3. Provide accurate information. If you need more clarification, always crosscheck the facts before posting.
4. Treat people online as you would treat them in real life.

You should not:

1. Make personal attacks on others.
2. Make false statements against others.
3. Bully or make inflammatory remarks towards other community members.
4. Use inappropriate language.
5. Post profanity or prejudiced comments.
6. Post sexually explicit content.
7. Post spam.

Failing to adhere to the above-mentioned rules may result in a permanent ban.

What should I do when I want to report such behavior?

1. Click on the flag icon below the offending post.
2. Select a category and click the "Flag Post" button.
3. A forum moderator will review the flagged content soon.

---

id: pips-forum title: Polygon Community Forum sidebar_label: Community Forum description: The Polygon Community Forum is the central hub of the Polygon ecosystem. keywords: - docs - matic - polygon - DAO - governance image: https://wiki.polygon.technology/img/polygon-logo.png

The Polygon Community Forum is the central hub of the Polygon ecosystem, where community members, developers, validators, and projects deployed on Polygon PoS come together to collaborate on protocol changes and share ideas and issues around the Polygon ecosystem and its various products.

Completely open, accessible, and transparent, the forum welcomes newcomers to join in building the future of scaling solutions for Ethereum. In addition to meeting other creative community members, the forum offers an excellent opportunity to learn about how different Polygon solutions are integrated into projects. It's also a great place to propose and discuss different types of Polygon Improvement Proposals, ranging from informational to core protocol changes.

If you haven't already, sign up here, and be sure to check out the next page for the community guidelines.

id: how-to-propose title: How to Propose description: Learn how to raise proposals in the PIP program. keywords: - docs - matic - polygon - ethereum image: https://wiki.polygon.technology/img/polygon-logo.png

Polygon Improvement Proposals, or PIPs, are a way for the Polygon community to propose and develop solutions to improve the Polygon protocols. PIPs are an important part of the Polygon development process and provide a way for developers, researchers, and users to contribute to the future of the Polygon network.

In this guide, we will cover the basics of how to write and submit a PIP for consideration. Whether you're a seasoned Polygon developer or just starting to explore blockchain development, this guide will provide you with the information and resources you need to get started with PIPs and make your voice heard in the Polygon community.

So before you begin crafting your proposal, ask yourself the following questions that will help you decide whether your idea is a PIP or a forum post!

Is it a PIP or an advertisment?

Do not use the PIP framework to promote your project!

Is the proposal unique?

Make sure that your proposed PIP is not a duplicate of a previous proposal.

Why are you writing a PIP?

A Polygon Improvement Proposal (PIP) is a document that outlines changes or improvements to the Polygon network.

PIPs can be written for a variety of valid reasons, including addressing technical issues, improving scalability, adding new features, enhancing security, and standardizing processes.

OK, I have checked the PIP framework, and consulted with the above questions. What now?

A Polygon Improvement Proposal (PIP) is a design document that proposes new features, improvements, or standards for the Polygon protocols. The following is a guide on how to write and submit a PIP:

1. Choose a topic: Choose a topic that you want to propose a change for. It could be a new feature, an improvement, or a novel standard.
2. Research and gather information: Gather information about the topic, including relevant background information, technical details, and any relevant existing solutions or standards.
3. Write the PIP: We suggest using the PIP template provided in the GitHub repository. The template includes sections for introduction, motivation, specification, and rationale.
4. Gather feedback: Share your PIP with the Polygon community on the Polygon community Forum and get feedback from experts, developers, and users. You can do this by sharing the PIP on forums, social media, or by reaching out to relevant individuals.
5. Submit the PIP: Submit the PIP to the Polygon Improvement Proposal repository on Github.
6. Revise and finalize: Revise your PIP based on the feedback you receive and finalize it.
7. Community review: The Polygon community will review the PIP and provide feedback. You may need to revise the PIP based on the feedback received.
8. Approval: If the PIP is approved, it will be included in the Polygon protocol.

Polygon Improvement Proposal (PIP) vs Polygon Request for Comments (PRC)

Polygon Improvement Proposal (PIP) and Polygon Request for Comments (PRC) are similar in that they both serve as a standard for proposing improvements and additions to the Polygon protocols. However, there is a subtle difference in the scope of the proposals they represent.

PIPs are broader in scope and can encompass proposals for improvements to the Polygon protocols, such as changes to the protocol, upgrades to the blockchain, and new

Here are some tips that will help you with moving the PIP forward!

- Use the Polygon Community Forum to engage with the community.
- Reach out to projects that are relevant to your proposal and ask them for feedback and ideas!

> *Tip: Under the Verified Group in the Polygon Forum you will find a wide range of dApps and projects deployed on Polygon PoS.*

- Write an article about the relevance of the PIP, explain your rationale, inspiration and how it can be used to improve the protocol/community.
- Engage in the discussion taking place in the Forum. Great minds work better together!

---

id: pips-overview title: Polygon Improvement Proposals (PIPs) sidebar_label: Overview description: Learn about Polygon Improvement Proposals. keywords: - docs - matic - polygon - DAO - governance image: https://wiki.polygon.technology/img/polygon-logo.png

import useBaseUrl from '@docusaurus/useBaseUrl';

What are PIPs?

Polygon Improvement Proposals (PIPs) describe standards for the Polygon ecosystem and govern the process through which the Polygon community introduces, finds consensus on, and implements changes to Polygon protocols. A PIP provides information to the Polygon community about suggested changes to the Polygon protocols. Authors use PIPs to document proposed changes, understand community sentiment, incorporate feedback, and provide technical specifications behind the suggested changes.

The PIP process

The Polygon Improvement Proposal (PIP) process is a preliminary approach for allowing the community to put forward-protocol upgrades that aim at improving Polygon PoS. This process is inspired by the PEP process, which has been used in the development of frameworks for other open-source projects, including Ethereum and Bitcoin. As a guiding rule, it is recommended that all changes originate from the Polygon Community Forum and the PIP GitHub repository in the form of a proposal written in line with PIP-1. It is suggested to consult with the Governance processes board regarding implementing different types of PIPs, as separate classifications of PIPs have other implementation flows.

Who can propose a PIP?

Anyone can propose a PIP.

Why submit a PIP?

- To introduce changes to the Polygon protocols.
- To offer new standards to the ecosystem.
- To encourage open and transparent discussion and decision-making about changes to the Polygon protocols, ensuring that any changes are well thought-out and aligned with the goals of the Polygon community.

Where are the PIPs hosted?

The PIPs are available on GitHub.

What should I do before submitting a PIP?

Before submitting a PIP, look through the PIP list and the Polygon Community Forum to see if someone has had a similar idea. If so, join the discussion. If there is a different proposal, submit the PIP yourself.

What do I need to do if I want to write a PIP?

Go to GitHub and use the suggested template to draft your PIP. After writing your proposal, follow the steps described in PIP-1 and PIP-8.

What does the PIP process look like?

Take a look at the flowchart that explains how the PIP process works.

What are the types of PIPs?

There are four types of PIPs that differ based on the changes the author wants to introduce. More specifically:

- The **Core** category is for changes to the Heimdall and Bor protocols, which are the backbone of the Polygon network. This category is for discussing, proposing, and implementing modifications to these protocols to improve the overall functioning of the Polygon network.

  Examples include but are not limited to:

  i. Proposal to upgrade Heimdall to a new version to improve security and scalability.

  ii. Discussion on changing the block size limit in Bor to accommodate more transactions (PIP-5 & PIP-6). iii. Implement a new consensus algorithm to improve the efficiency of the Polygon network.

- The **Contracts** category is for changes to the Core contracts deployed on the Ethereum network. This includes discussing and proposing changes to the smart contracts that run on the Ethereum network and changes to the security and functioning of these contracts.

  Examples include but are not limited to:

  i. Proposal to add new functionality to the Core staking contracts.

  ii. Discussions on changes related to bridge security.

- The **Interface** category is for client API/RPC specifications and standards improvements. This includes changes to the way that clients interact with the Polygon network, as well as changes to the way that data is transmitted and received by these clients.

  Examples include but are not limited to:

  i. Contract standards - PRCs.

ii. Proposal to improve the API/RPC specification for transmitting data between clients and the Polygon network. iii. Discussion on changes to the way clients interact with the Polygon network to make it more user-friendly. iv. Implement a new data format for transmitting information between clients and the Polygon network.

- The **Informational** category is for issues and ideas that the Polygon community can adopt. This includes discussions and proposals on new features, changes to existing features, and best practices for using the Polygon network. This category is meant to be a place for members of the Polygon community to share information and collaborate on new ideas and improvements to the network.

  Examples include but are not limited to:

  i. Sharing information and resources on integrating a specific type of Dapp to the Polygon protocols.

  ii. Discussion on best practices and risk management for Dapps.

What is the difference between a PIP and a PRC, and how to decide which one I am submitting?

There is little of a difference.

Polygon Improvement Proposal (PIP) and Polygon Request for Comments (PRC) are similar in that they both serve as a standard for proposing Polygon protocol improvements and additions. However, the scope of the proposals they represent differs slightly.

PIPs have a broader scope and can include proposals for Polygon protocol improvements such as protocol changes, blockchain upgrades, and new features.

Conversely, PRC is primarily concerned with proposals for standardizing smart contracts based on the Polygon protocols. PRCs define smart contract development and deployment standards, such as token standards, and serve as a foundation for developers to build on.

In summary, PIPs are for proposals that aim to improve the Polygon protocols as a whole, while PRCs are for proposals that aim to standardize smart contract development on Polygon protocols.

If you are still determining which category your proposal falls under, don't worry; PIP Editors can help you with that.

Where can I see the proposed PIPs?

- Polygon Community Forum
- Github page
- Official PIP Website (soon 👀)

Where do PIP-related discussions take place?

- Polygon Community Forum
- Polygon Protocol Governance Calls

What to do after submitting a PIP to the PIP GitHub repository?

- Share the PIP with the community at the Polygon Community Forum. Present your PIP as submitted, and focus your discussion by highlighting why it should be implemented. Why is your PIP different from other similar PIPs or PRCs? What are the benefits of adopting your proposal?
- Once doing so, remember to engage in the discussions. Address the comments posted and (if applicable) incorporate them in the following versions of the PIP.

How long does the whole process of a PIP, from submission to finalization, take?

- The timeline may vary depending on the community's response and the author's responsiveness to the comments received.

My PIP is still in draft. What to do to make it final?

- Moving the PIP through the different stages is the responsibility of the author(s). Once you are happy with the state of the proposal, you should move the proposal to **Review** and start receiving comments from the community.

Who are the PIP Editors, and what do they do?

A PIP editor's role is to facilitate the PIP process, and they DON'T hold the authority to decide if the PIP is finally approved (whether by on-chain consensus or ecosystem adoption).

PIP Editors ensure that:

1. PIP is well formatted and technically complete.
2. The PIP repository is maintained, and the standards for PIPs are met.

How to become a PIP Editor, and what does the selection process look like?

Everyone can apply to become a PIP editor, given they match the following requirements.

- Understands Polygon protocols at an expert level.
- Is familiar with the PIP and similar frameworks used on other decentralized blockchains.
- Is proficient in using GitHub.
- Is a good communicator.
- Can appropriately handle contentious discourse.

Read more about it here.

id: bor-chain title: What is BoR-Chain? sidebar_label: Bor Chain description: Introduction to Bor Chain keywords: - docs - matic - polygon - bor chain image: https://wiki.polygon.technology/img/polygon-logo.png

The Bor node, or the block producer implementation, is an EVM-compatible blockchain operator. Currently, it is a basic Geth implementation with custom changes done to the consensus algorithm. However, this will be built from the ground up to make it lightweight and focused.

Block producers are chosen from the validator set and are shuffled using historical Ethereum block hashes for the same purpose. However, we are exploring sources of randomness for this selection.

id: staking title: Staking on Polygon description: Staking on Polygon keywords: - docs - polygon - matic - staking - unstake - restake image: https://wiki.polygon.technology/img/polygon-logo.png

Staking on Polygon

For the Polygon Network, any participant can be qualified to become a network validator by running a full node. The primary incentive for running a full node for validators is to earn Rewards and Transaction fees. Validator participating in consensus for Polygon incentivise to participate as they receives block rewards and transaction fee.

As validator slots are limited for the network, the process to get selected as a validator is to participate in an on-chain auction which happens at regular intervals as defined here.

Stake

If the slot is open, then the auction is started to interested validators:

- Where they will bid more than the last bid made for the slot.
- The process of participating in auction is outlined here:
  - Auction is automatically started once the slot is opened.
  - To start participating in auction, call `startAuction()`
  - This will lock your assets in Stack Manager.
  - If another potential validator stakes more than your stake, then locked tokens will be returned back to you.
  - Again, stake more to win the auction.
- At the end of the auction period, the highest bidder wins and becomes a Validator on the Polygon network.

:::note

Please keep your full node running if you are participating in the auction.

:::

The process of becoming a validator after the highest bidder wins the slot is outlined below:

- Call `confirmAuction()` to confirm your participation.
- Bridge on Heimdall listens to this event and broadcasts to Heimdall.
- After consensus, validator is added to Heimdall but not activated.
- Validator starts validating only after `startEpoch` (defined here).
- As soon as `startEpoch` reaches, the validator is added to `validator-set` and starts participating in the consensus mechanism.

:::info Recommended

To ensure security for validators stake, we recommend validators to provide different `signer` address from which verification of `checkPoint` sigs will be handled. This is to keep signing key separate from validator's wallet key so that funds are protected in case of a node hack.

:::

UnStake

Unstaking allows validator to be out of the active pool of validators. In order to ensure **Good Participation**, their stake is locked for the next 21 days.

When validators want to exit from the network and stop validating blocks and submitting checkpoints, they can `unstake`. This action is immediate as of now. After this action, validator is considered out from the active set of validators.

ReStake

Validators can also add more stake into their amount in order to earn more rewards and be competitive for their validator spot and maintain their position.

id: heimdall-chain title: What is Heimdall Chain? sidebar_label: Heimdall Chain description: Build your next blockchain app on Polygon. keywords: - docs - matic - polygon - heimdall - checkpoint - pos - verifier image: https://wiki.polygon.technology/img/polygon-logo.png

Heimdall Chain

Heimdall is Polygon Proof-of-Stake Verifier layer, which is responsible for checkpointing a representation of blocks to the main chain in our architecture. We have implemented this by building on top of the Tendermint consensus engine with changes to the signature scheme and various data structures.

The main chain Stake Manager contract works in conjunction with the Heimdall node to act as the trustless stake management mechanism for the PoS engine, including selecting the validator set, updating validators, etc. Since staking is actually done on the Ethereum smart contract, we do not rely only on validator honesty and instead inherit Ethereum chain security for this key part.

Heimdall layer handles the aggregation of blocks produced by Bor into a merkle tree and publishing the merkle root periodically to the root chain. This periodic publishing is called **"checkpoint"**. For every few blocks on Bor, a validator (on the Heimdall layer):

1. Validates all the blocks since the last checkpoint
2. Creates a merkle tree of the block hashes
3. Publishes the merkle root to the main chain

Checkpoints are important for two reasons:

1. Providing finality on the Root Chain
2. Providing Proof of Burn in withdrawal of assets

A bird's eye view of the process can be explained as:

- A subset of active validators from the pool are selected to act as block producers for a span. The selection of each span will also be consented by at least 2/3 in power. These block producers are responsible for creating blocks and broadcasting them to the remaining network.
- A checkpoint includes the root of all blocks created during any given interval. All nodes validate the same and attach their signatures to it.
- A selected proposer from the validator set is responsible for collecting all signatures for a particular checkpoint and committing the same on the main chain.
- The responsibility of creating blocks and also proposing checkpoints is variably dependent on a validator's stake ratio in the overall pool.

---

id: polygon-architecture title: Polygon PoS Architecture description: Polygon PoS Architecture including Heimdall and Bor chains keywords: - docs - matic - polygon - architecture - pos - blockchain image: https://wiki.polygon.technology/img/polygon-logo.png

import useBaseUrl from '@docusaurus/useBaseUrl';

Polygon PoS Architecture

Polygon PoS mainnet is a blockchain application platform. Polygon PoS features a generic validation layer separated from varying execution environments like full-blown EVM sidechains and other layer 2 approaches such as zero-knowledge rollups.

To enable the PoS mechanism on our platform, a set of **staking** management contracts are deployed on Ethereum, as well as a set of incentivized validators running **Heimdall** and **Bor** nodes. Ethereum is the first basechain Polygon supports, but Polygon intends to offer support for additional basechains, based on community suggestions and consensus, to enable an interoperable decentralized Layer 2 blockchain platform.

Polygon PoS has a three-layer architecture:

1. Staking smart contracts on Ethereum
2. Heimdall (Proof of Stake layer)
3. Bor (Block producer layer)

<img src={useBaseUrl("img/matic/Architecture.png")} />

Polygon smart contracts (on Ethereum)

Polygon maintains a set of smart contracts on Ethereum, which handle the following:

- Staking management for the Proof-of-Stake layer
- Delegation management including validator shares
- Checkpoints/snapshots of state

Heimdall (Proof-of-Stake validator layer)

Heimdall is the PoS validator node that works in consonance with the staking contracts on Ethereum to enable the PoS mechanism on Polygon. We have implemented this by building on top of the Tendermint consensus engine with changes to the signature scheme and various data structures. It is responsible for block validation, block producer committee selection, checkpointing a representation of the blocks to Ethereum in our architecture and various other responsibilities.

Heimdall layer handles the aggregation of blocks produced by Bor into a merkle tree and publishing the merkle root periodically to the root chain. These periodic publishings are called `checkpoints`. For every few blocks on Bor, a validator (on the Heimdall layer):

1. Validates all the blocks since the last checkpoint
2. Creates a merkle tree of the block hashes
3. Publishes the merkle root to the main chain

Checkpoints are important for two reasons:

1. Providing finality on the Root Chain
2. Providing Proof of Burn in withdrawal of assets

A bird's eye view of the process can be explained as:

- A subset of active validators from the pool are selected to act as block producers for a span. The Selection of each span will also be consented by at least 2/3 in power. These block producers are responsible for creating blocks and broadcasting it to the remaining network.
- A checkpoint includes the root of all blocks created during any given interval. All nodes validate the same and attach their signature to it.

- A selected proposer from the validator set is responsible for collecting all signatures for a particular checkpoint and committing the same on the main chain.
- The responsibility of creating blocks and also proposing checkpoints is variably dependent on a validator's stake ratio in the overall pool.

Bor (Block Producer Layer)

Bor is Polygon block producer layer - the entity responsible for aggregating transactions into blocks.

Block producers are periodically shuffled via committee selection on Heimdall in durations termed as a `span` in Polygon. Blocks are produced at the **Bor** node and the VM is EVM-compatible. Blocks produced on Bor are also validated periodically by Heimdall nodes, and a checkpoint consisting of the Merkle tree hash of a set of blocks on Bor is committed to Ethereum periodically.

Further Reading

- Building with Polygon Node Provider
- Deep Dive into Polygon Architecture

Resources

- Bor Architecture
- Heimdall Architecture
- Checkpoint Mechanism

---

id: matic-flow title: How Polygon Works description: Build your next blockchain app on Polygon. keywords: - docs - matic - polygon - how polygon works image: https://wiki.polygon.technology/img/polygon-logo.png

import useBaseUrl from '@docusaurus/useBaseUrl';

How Polygon Works

Polygon has a three-layer architecture:

1. Staking smart contracts on Ethereum
2. Heimdall (Proof of Stake layer)
3. Bor (Block producer layer)

The below image will help you understand how these core components interact with each other:

<img src={useBaseUrl("img/Bor/bor-architecture.png")} />

id: security-models title: Proof of Stake Security Model sidebar_label: Security Model description: "Learn about the PoS security model." keywords: - docs - matic - polygon - security - implementation image: https://wiki.polygon.technology/img/polygon-logo.png

Overview

The Proof of Stake (PoS) model's security relies on the Heimdall and Bor layers, both constructed upon the Tendermint protocol. A checkpoint only commits to the root chain once a minimum of two-thirds of the validators provide their signatures.

The PoS mechanism operates through a collection of staking management contracts on Ethereum, supplemented by a group of incentivized validators running Heimdall and Bor nodes. This approach enables:

Participation as a Validator by staking MATIC tokens on the Ethereum smart contract. Earning of staking rewards for validating state transitions on the Polygon network. A fast finality layer periodically solidifies the state through checkpoints, enhancing state security. The EVM-compatible chain, with its high throughput and swift block time, optimizes scalability over an extensive degree of decentralization. Heimdall ensures a secure final state commit, passing through a large validator set to maintain high decentralization.

For Developers

Deploying on the PoS security model is streamlined for dApp developers: simply deploy the smart contract on the Polygon PoS network. This ease of deployment is due to the account-based architecture, which is compatible with the EVM blockchain.

id: setup-bor title: Setup Bor description: Setup Bor node keywords: - docs - matic - polygon - setup bor image: https://wiki.polygon.technology/img/polygon-logo.png

Setup Bor

Use the `master` or `develop` branch, which contains the latest stable release.

```
$ mkdir -p $GOPATH/src/github.com/maticnetwork
$ cd $GOPATH/src/github.com/maticnetwork
$ git clone https://github.com/maticnetwork/bor
```

```
$ cd bor
$ make bor-all
```

Now, you have Bor installed on your local system and the binary is available in the path `./build/bin/bor`.

Connecting to console (optional)

This is an optional step. You need not connect to a console. You can do so only if you are interested in other details.

Just like Geth, you can connect to bor console to execute various types of queries. From your `dataDir`, run the following command:

```
$ cd ~/matic/tesnets
$ git submodule init
$ git submodule update

$ cd ~/matic/tesnets/genesis-contracts
$ npm install

$ git submodule init
$ git submodule update
$ cd ~/matic/tesnets/genesis-contracts/matic-contracts
$ npm install
$ node scripts/process-templates.js --bor-chain-id 15001
$ npm run truffle:compile
```

Once templates are processed, we need to set validators in `tesnets/genesis-contracts/validators.js` file. This file should look like this:

```
const validators = [
  {
    address: "0x6c468CF8c9879006E22EC4029696E005C2319C9D",
    stake: 10, // without 10^18
    balance: 1000 // without 10^18
  }
]
```

Generate Bor validator set using `validators.js` file:

```
$ cd ~/matic/testnets/genesis-contracts
$ node generate-borvalidatorset.js --bor-chain-id 15001 --heimdall-chain-id heimdall-P5rXwg
```

This command will generate `genesis-contracts/contracts/BorValidatorSet.sol`.

Generate genesis.json, once `BorValidatorSet.sol` is generated:

```
$ cd ~/matic/testnets/genesis-contracts
$ node generate-genesis.js --bor-chain-id 15001 --heimdall-chain-id heimdall-P5rXwg
```

This will generate `genesis-contracts/genesis.json`.

Start Bor

Once genesis file is generated at `~/matic/tesnets/genesis-contracts/genesis.json`, prepare Bor node:

```
$ cd ~/matic/testnets/bor-devnet
$ bash setup.sh
```

Start Bor using the following command:

```
$ cd ~/matic/testnets/bor-devnet
$ bash start.sh 1
```

Bor will start running at port 8545.

If you want to clean Bor and start again:

```
$ bash clean.sh
$ bash setup.sh
$ bash start.sh 1
```

Test Bor and Heimdall

To test both Bor and Heimdall, you need to run Bor and Heimdall, Heimdall's rest-server, and Bridge in parallel.

Run Heimdall rest-server (optional)

Follow this [guide]() instructions to run nginx on your local machine (Mac OSX).

Add below content into `/usr/local/etc/nginx/nginx.conf` and restart nginx:

```
    worker_processes  1;

    events {
        worker_connections 1024;
    }

    http {
        server {
            listen 80;
            server_name localhost;

            location / {
              add_header 'Access-Control-Allow-Origin' * always;
              add_header 'Access-Control-Allow-Credentials' 'true';
              add_header 'Access-Control-Allow-Headers' 'Authorization,Accept,Origin,DNT,X-CustomHeader,Keep-Alive,User-Agent,X-
Requested-With,If-Modified-Since,Cache-Control,Content-Type,Content-Range,Range';
              add_header 'Access-Control-Allow-Methods' 'GET,POST,OPTIONS,PUT,DELETE,PATCH';

              if ($request_method = 'OPTIONS') {
                add_header 'Access-Control-Allow-Origin' * always;
                add_header 'Access-Control-Allow-Credentials' 'true';
                add_header 'Access-Control-Allow-Headers' 'Authorization,Accept,Origin,DNT,X-CustomHeader,Keep-Alive,User-
Agent,X-Requested-With,If-Modified-Since,Cache-Control,Content-Type,Content-Range,Range';
                add_header 'Access-Control-Allow-Methods' 'GET,POST,OPTIONS,PUT,DELETE,PATCH';
                add_header 'Access-Control-Max-Age' 1728000;
                add_header 'Content-Type' 'text/plain charset=UTF-8';
                add_header 'Content-Length' 0;
                return 204;
              }

              proxy_redirect off;
              proxy_set_header host $host;
              proxy_set_header X-real-ip $remote_addr;
              proxy_set_header X-forward-for $proxy_add_x_forwarded_for;
              proxy_pass http://127.0.0.1:1317;
            }
        }
    }
```

Reload nginx using new config changes:

```
    sudo nginx -s reload
```---
id: new-to-polygon
title: Welcome to Polygon
description: Build your next blockchain application on the Polygon network.
keywords:
  - docs
  - polygon
  - new to polygon
  - wiki
image: https://wiki.polygon.technology/img/polygon-logo.png

Polygon is a scaling solution for public blockchain networks. Polygon PoS chain supports all the existing Ethereum tooling along with faster and cheaper transactions.

What is a Blockchain?

Blockchain is a shared, immutable ledger for recording transactions, tracking assets and building trust. Head over to [Blockchain Basics](blockchain-basics/basics-blockchain.md) section to read more.

Types of Interaction on Polygon

- [Polygon PoS chain](/develop/getting-started.md)
- [Ethereum + Polygon with PoS bridge](/develop/ethereum-polygon/pos/getting-started.md)

Query the Blockchain

Most blockchain interactions involve reading its state.

Alchemy offers a reference guide on how to make basic requests to the blockchain. Check out their guide on [how to query Polygon](https://docs.alchemy.com/reference/polygon-sdk-examples).

```
## Deploy Smart Contracts

- Deploy your Contracts on Polygon
  - [Using thirdweb](/develop/thirdweb.md)
  - [Using Alchemy](/develop/alchemy.md)
  - [Using QuickNode](/develop/quicknode.md)
  - [Using Chainstack](/develop/chainstack.md)
  - [Using Remix](/develop/remix.md)
  - [Using Truffle](/develop/truffle.md)
  - [Using Hardhat](/develop/hardhat.md)

:::note

Configure the Web3 RPC-URL to "https://rpc-mumbai.matic.today", everything else remains the same.

:::

## Validator and delegator roles

On the Polygon Network, you can be a validator or a delegator. See:

- [Who Is a Validator](/maintain/polygon-basics/who-is-validator.md)
- [Who Is a Delegator](/maintain/polygon-basics/who-is-delegator.md)

## Architecture

If your goal is to become a validator, it is essential that you understand the Polygon architecture.

See [Polygon Architecture](/pos/design/validator/architecture.md).

### Components

To have a granular understanding of the Polygon architecture, see core components:

- [Heimdall](/pos/design/heimdall/overview.md)
- [Bor](/pos/design/bor/overview.md)
- [Contracts](/pos/contracts/stakingmanager.md)

### Codebases

To have a granular understanding of the core components, check out their codebases:

- [Heimdall](https://github.com/maticnetwork/heimdall)
- [Bor](https://github.com/maticnetwork/bor)
- [Contracts](https://github.com/maticnetwork/contracts)

## How-Tos

### Node setup

If you want to run a full node on the Polygon Mainnet or Mumbai Testnet, you can follow the
[Run a Validator Node](/pos/operate/validator/run-validator.md) guide.

### Staking operations

- [Validator Staking Operations](/docs/pos/operate/validator/validator-staking-operations)
- [Delegate](/docs/maintain/delegate/delegate)

### External Resources

- [Your first dApp](https://www.youtube.com/watch?v=rzvk2kdjr2I)
- [Sidechains and Childchains](https://hackernoon.com/what-are-sidechains-and-childchains-7202cc9e5994)
---
id: glossary
title: Glossary
description: Key Polygon terms
keywords:
  - docs
  - matic
  - polygon
  - glossary
  - jargons
slug: glossary
image: https://wiki.polygon.technology/img/polygon-logo.png
---

## Block Producer
```

A block producer is an active [validator](#validator) selected to act as block producer for a [span](#span).

A block producer is responsible for creating blocks and broadcasting the created blocks to the network.

Bor

A Bor node is a node producing blocks on the Polygon Network.

Bor is based on [Go Ethereum](https://geth.ethereum.org/).

Checkpoint Transaction

A checkpoint transaction is a transaction containing the Merkle root of blocks of the [Bor](#bor) layer between the checkpoint intervals.

The transaction is committed to the Polygon staking contracts on the Ethereum mainnet by a [Heimdall](#heimdall) node.

See also:

* [Heimdall architecture: Checkpoint](/docs/pos/design/heimdall/checkpoint)
* [Checkpoint Mechanism](/docs/pos/design/validator/core-components/checkpoint-mechanism)

Commission

A commission is the percentage of the rewards taken by [validators](#validator) from the [delegators](#delegator) who stake with the validators.

See also [Validator Commission Operations](/docs/pos/operate/validator/validator-commission-operations).

Delegator

The delegator role stakes the MATIC tokens to secure the Polygon Network with existing [validators](#validator) without running the nodes themselves.

See also [Who Is a Delegator](/docs/maintain/polygon-basics/who-is-delegator).

Full Node

A full node is a fully synced [sentry node](#sentry) running both [Heimdall](#heimdall) and [Bor](#bor).

See also [Full Node Deployment](/docs/operate/full-node-deployment).

Heimdall

A Heimdall node is a node running in parallel to the Ethereum mainnet, monitoring the set of contracts deployed on the Ethereum mainnet, and committing the Polygon Network [checkpoints](#checkpoint-transaction) to the Ethereum mainnet.

Heimdall is based on [Tendermint](https://tendermint.com/).

Owner Address

An owner address is the address used to stake, restake, change the signer address, withdraw rewards and manage delegation related parameters on the Ethereum mainnet.

While the [signer key](#signer-address) is kept on the node and is considered a **hot** wallet, the owner key must be kept very secure, used infrequently, and is considered a **cold** wallet.

See also [Key Management](validator/core-components/key-management.md).

Proposer

A proposer is the [validator](#validator) selected by the algorithm to propose a new block.

A proposer is also responsible for collecting all signatures for a particular [checkpoint](#checkpoint-transaction) and committing the checkpoint to the Ethereum mainnet.

Sentry

A sentry node is a node running both the [Heimdall](#heimdall) node and the [Bor](#bor) node to download the data from other nodes on the network and to propagate the [validator](#validator) data on the network.

A sentry node is open to all other sentry nodes on the network.

Span

A logically defined set of blocks for which a set of validators is chosen from all the available [validators](#validator).

The selection of each span is decided by at least 2/3 of the validators in terms of the staking power.

See also [Bor Consensus: Span](/docs/pos/design/bor/consensus.md#span).

Staking

Staking is the process of locking up tokens into a deposit to earn the right to validate and produce blocks on a blockchain. Typically staking is done in the native token for the network — for the MATIC token is locked up by validators / stakers in the Polygon Network. Other examples include ETH in Ethereum (post-merge), ATOM in Cosmos, etc.

See also [What Is Proof of Stake](polygon-basics/what-is-proof-of-stake.md).

Signer address

A signer address is the address of an Ethereum account of the [Heimdall](#heimdall) validator node. The signer address signs and submits the [checkpoint transactions](#checkpoint-transaction).

While the signer key is kept on the node and is considered a **hot** wallet, the [owner key](#owner-address) must be kept very secure, used infrequently, and is considered a **cold** wallet.

See also [Key Management](validator/core-components/key-management.md).

Validator

Validators [stake their MATIC tokens](/docs/pos/operate/validator/validator-staking-operations) via staking contracts deployed on the Ethereum mainnet and are running both the [Heimdall](#heimdall) node and the [Bor](#bor) node to commit the network checkpoints to the Ethereum mainnet and to produce blocks on the network.

A validator node is only open to its [sentry](#sentry) node and closed to the rest of the network.

See also [Who Is a Validator](polygon-basics/who-is-validator.md).
---
id: liquid-delegation
title: Liquid Delegation
sidebar_label: Liquid Delegation
description: How Polygon uses liquid delegation to maintain the network.
keywords:
  - docs
  - polygon
  - matic
  - delegation
  - liquid delegation
slug: liquid-delegation
image: https://wiki.polygon.technology/img/polygon-logo.png

In a traditional Proof of Stake mechanism, the blockchain keeps track of a set of validators. Anyone can join this rank or right to validate transactions by sending a special type of transaction which stakes their coins(in Ethereum's case, ETH) and locks up into a deposit. Afterwards, the process of creating and agreeing to new blocks is done through a consensus algorithm by all active validators.

They lock up part of their stake for a certain amount of time (like a security deposit), and in return they get a chance proportional to that stake to select the next block.

Staking rewards are distributed as an incentive to the participants.

Delegation

Staking may be expensive, raising the barrier to entry, which favours the rich getting richer. Everyone should take part in network security and receive tokens of appreciation. The only other option is to join a staking pool similar to a mining pool, where validators must be trusted. We believe that sticking to the protocol is the best course of action for new delegators. Since capital and rewards are open and protected by in-protocol mechanisms.

Delegators can take part in validation even though they don't host entire nodes. However, by staking with validators, they can increase the network's strength and gain rewards by paying a tiny commission charge (which varies depending on the validator) to the validator of their choice.

Limitation of Traditional Delegator and Validator

Capital lockup cost for both validators and delegators is high due to Proof of Stake protocol design.

Still we can bring more liquidity view mechanism like validator NFT where any new party who wants to become a validator can buy validator NFT from a validator who wants to exit from system for some reason.

In case of delegators the amount locked is assumed to be in smaller chunks so we want that to be liquid so that participation is more active (i.e. if some delegator thinks that right now opportunities are great in DeFi but their capital is locked in staking pool even for withdrawal, they still need to wait for 21 days).

Also, locking up X ETH in a deposit is not free; it entails a sacrifice of optionality for the ETH holder. Right now, if you have 1000 ETH, you can do whatever you want with it. If you lock it up in a deposit, then it's stuck there for months in order to prevent attacks like [**nothing at stake**](https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQ#what-is-the-nothing-at-stake-problem-and-how-can-it-be-fixed) and punish validators for their bad participation.

In-Protocol vs Application Layer

Application-level staking liquidation has trust problem. Protocol-level staking liquidation is much more appreciated due to the fact that any new actor can trust it (which attracts more capital, even from smaller actors / delegators).

Polygon's Solution for Delegation

While exploring delegation, we realised that delegation needs to be in-protocol in order to have more trust from delegators.

We were facing similar issue to validators capital liquidity and thought of making it a NFT which can be transfers and exploring on similar thoughts like how it can be made more liquid and sikka-chorus.one 's [awesome design](https://blog.chorus.one/delegation-vouchers/) came to attention.

Thinking in terms of making is share of validator pool is great idea and since Polygon's staking is implemented on Ethereum smart contract it opens up a lot more options for us like making it ERC20 compatible so that it can be used in defi protocols.

As of now each validator has their own VMatic (i.e. for validator Ashish there will be AMatic token) because each validator has different performance (rewards and commission rate). Delegators can buy multiple validator share and hedge their risk towards poor performance of particular validator.

Advantages

- Since our design follows ERC20 like interface in delegation implementation, DeFi applications can be easily built on top of it.
- Delegated tokens can be used in lending protocols.
- Delegators can hedge their risk via prediction markets like Auger.

Future scope:

- Currently ERC20 are not fungible with other validators ERC20 / Share tokens but in future we think many new DeFi applications can build upon it and make some markets for it or even some better products.
- With chorus.one initiated research, we are also exploring problems like validators shorting their own tokens and other problems (shorting problems can be avoided via things like validator locking their own stake for X months and other things like validator insurance (on-chain) which will bring more trust for delegators).
- Delegator voting rights in order to participate in governance decisions.
- While making delegation liquid, we also want to ensure network security. That's why, in some form, slash-able capital is locked in case of fraud activity.

Given above design available in-protocol, validators can always implement their own similar mechanisms and stake via a contract which won't be available in Polygon Staking UI.

Future Goals

Things like interchain / cross-chain via Cosmos hub and Everett B-harvest design.

Resources

- [Vitalik's pos design](https://medium.com/@VitalikButerin/a-proof-of-stake-design-philosophy-506585978d51)
- [Intro to Staking Derivatives](https://medium.com/lemniscap/an-intro-to-staking-derivatives-i-a43054efd51c)
- [Staking Pools](https://slideslive.com/38920085/ethereum-20-trustless-staking-pools)
- [Inflation in Proof of Stake](https://medium.com/figment-networks/mis-understanding-yield-and-inflation-in-proof-of-stake-networks-6fea7e7c0e41)

```
---
id: what-is-proof-of-stake
title: What is Proof of Stake?
description: Learn what is Proof of Stake consensus mechanism
keywords:
  - docs
  - matic
  - polygon
  - stake
  - delegate
  - validate
  - pos
image: https://wiki.polygon.technology/img/polygon-logo.png
---
```

What is Proof of Stake?

Proof of Stake (PoS) is a category of consensus algorithms for public blockchains that depend on a validator's economic [stake](/docs/maintain/glossary.md#staking) in the network.

In Proof of Work (PoW) based public blockchains, the algorithm rewards participants who solve cryptographic puzzles to validate

transactions and create new blocks. PoW blockchain examples: Bitcoin, Ethereum (before merge).

In PoS-based public blockchains, a set of validators take turns proposing and voting on the next block. The weight of each validator's vote depends on the size of its deposit — [stake](/docs/maintain/glossary.md#staking). Significant advantages of PoS include security, reduced risk of centralization, and energy efficiency. PoS blockchain examples: Ethereum 2.0, Polygon.

In general, a PoS algorithm looks as follows. The blockchain keeps track of a set of validators, and anyone who holds the blockchain's base cryptocurrency (in Ethereum's case, ETH) can become a validator by sending a special type of transaction that locks up their ETH into a deposit. The process of creating and agreeing to new blocks is then done through a consensus algorithm that all current validators can participate in.

There are many kinds of consensus algorithms, and many ways to assign rewards to validators who participate in the consensus algorithm, so there are many "flavors" of Proof of Stake. From an algorithmic perspective, there are two major types: chain-based PoS and [BFT-style PoS](https://en.wikipedia.org/wiki/Byzantine_fault_tolerance).

In **chain-based Proof of Stake**, the algorithm pseudo-randomly selects a validator during each time slot (e.g. every period of 10 seconds might be a time slot), and assigns that validator the right to create a single block, and this block must point to some previous block (normally the block at the end of the previously longest chain), and so over time most blocks converge into a single constantly growing chain.

In **BFT-style Proof of Stake**, validators are **randomly** assigned the right to **propose** blocks. Agreement on which block is **canonical** is done through a multi-round process where every validator sends a **Vote** for some specific block during each round, and at the end of the process, all (honest and online) validators permanently agree on whether or not any given block is part of the chain. Note that blocks may still be **chained together**. The key difference is that consensus on a block can come within one block, and does not depend on the length or size of the chain after it.

For more details, refer to https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQ.

See Also

* [Delegator](/docs/maintain/glossary.md#delegator)
* [Validator](/docs/maintain/glossary.md#validator)

---
id: who-is-delegator
title: Who is a Delegator
description: Token holders who do not run a node
keywords:
  - docs
  - matic
  - polygon
  - delegator
  - Who is a Delegator
image: https://wiki.polygon.technology/img/polygon-logo.png

Delegators are token holders who cannot, or do not want to run a [validator](/docs/maintain/glossary.md#validator) node themselves. Instead, they secure the network by delegating their stake to validator nodes and play a critical role in the system, as they are responsible for choosing validators. They run their delegation transaction on the staking contract on the Ethereum mainnet.

The MATIC tokens are bonded with the next [checkpoint](/docs/maintain/glossary.md#checkpoint-transaction) committed on the Ethereum mainnet. Delegators also have an option to opt out of the system whenever they want. Similar to validators, delegators have to wait for the unbonding period, which consists of approximately 9 days, to end before withdrawing their stake.

Fees and Rewards

Delegators stake their tokens by delegating them to validator, obtaining a percentage of their rewards in exchange. Because delegators share rewards with their validators, delegators also share risks. Should a validator misbehave, each of their delegators are at risk of being partially slashed in proportion to their delegated stake.

Validators set a [commission](/docs/maintain/glossary.md#commission) percentage to determine the percentage of rewards that will go to them. Delegators are able to view the commission rate of each validator to understand each validator's reward distribution and a relative rate of return on their stake.

:::caution Validators with a 100% commission rate

These are validators who take all of the rewards and are not looking for delegation, as they have enough tokens to stake on their own.

:::

Delegators have the option to re-delegate their tokens to other validators. Rewards are accumulated at every checkpoint.

:::tip Being an active delegator

Delegation should not be seen as a passive activity, as delegators are an integral part of maintaining the Polygon network. Each delegator is responsible for managing their own risk, but in doing so, delegators

should aim to elect validators that are behaving well.

:::

See Also

* [Delegate](/docs/maintain/delegate/delegate)
* [Validator FAQ](/docs/pos/operate/validator/faq/validator-faq)

---
id: who-is-validator
title: Who is a Validator
sidebar_label: Who is a Validator
description: "A participant in the network who runs Heimdall and Bor nodes."
keywords:
  - docs
  - matic
  - polygon
  - validator
  - Who is a Validator
image: https://wiki.polygon.technology/img/polygon-logo.png

Validator is a participant in the network who locks up MATIC tokens in the system and runs Heimdall validator and Bor block producer nodes in order to help run the network. Validators stake their MATIC tokens as collateral to work for the security of the network and in exchange for their service, earn rewards.

Rewards are distributed to all stakers proportional to their stake at every checkpoint with the exception being the proposer getting an additional bonus. User reward balance gets updated in the contract which is referred to while claiming rewards.

:::tip

Those who are interested in securing the network but are not running a full node can participate as [delegators](/docs/maintain/glossary.md#delegator).

:::

Overview

Validators participate as block producers and verifiers. Once a [checkpoint](/docs/maintain/glossary.md#checkpoint-transaction) is validated by the participants, updates are made on the parent chain (the Ethereum mainnet) which releases the rewards for validators depending on their stake in network.

Polygon relies on a set of [validators](/docs/maintain/glossary.md#validator) to secure the network. The role of validators is to run a full node, [produce blocks](/docs/maintain/glossary.md#block-producer), validate and participate in consensus, and commit [checkpoints](/docs/maintain/glossary.md#checkpoint-transaction) on the Ethereum mainnet. To become a validator, one needs to [stake](/docs/maintain/glossary.md#staking) their MATIC tokens with staking management contracts residing on the Ethereum mainnet.

Core components

[Heimdall](/docs/maintain/glossary.md#heimdall) reads the events emitted by the staking contracts to pick the validators for the current set with their updated stake ratio, which is used also by [Bor](/docs/maintain/glossary.md#bor) while producing blocks.

[Delegation](/docs/maintain/glossary.md#delegator) is also recorded in the staking contracts and any update in the validator power or node [signer address](/docs/maintain/glossary.md#signer-address) or unbonding requests comes into effect when the next checkpoint gets committed.

End-to-end flow for a Polygon Validator

Validators set up their signing nodes, sync data and then stake their tokens on the Ethereum mainnet staking contracts to be allocated a slot in the current set.

:::warning

At present, there are no open validator slots available on Polygon PoS. There is also a waitlist to become a validator. In the future, if slots become available, validators may apply to be considered and removed off of the waitlist.

:::

Block producers are chosen from the validator set where it is the responsibility of the selected validators to produce blocks for a given [span](/docs/maintain/glossary.md#span).

Nodes at Heimdall validate the blocks being produced, participate in consensus and commit checkpoints on the Ethereum mainnet at defined intervals.

The probability of validators to get selected as the block producer or checkpoint [proposer](/docs/maintain/glossary.md#proposer) is dependent on one’s stake ratio including delegations in the overall pool.

Validators receive rewards at every checkpoint as per their stake ratio, after deducting the proposer bonus which is disbursed to the checkpoint proposer.

One can opt out of the system at any time and can withdraw tokens once the unbonding period ends.

Economics

See [Rewards](/docs/pos/design/validator/rewards).

Setting up a Validator node

See [Validate](/docs/pos/operate/validator/validator-index).

See Also

* [Validator Responsibilities](/docs/pos/operate/validator/validator-responsibilities)
* [Validate](/docs/pos/operate/validator/validator-index)
* [Validator FAQ](/docs/pos/operate/validator/faq/validator-faq)

---
id: bitwise
title: Bitwise Chiplet
sidebar_label: Bitwise Chiplet
description: "How to compute bitwise operations on 32-bit values."
keywords:
  - docs
  - matic
  - polygon
  - wiki
  - miden
  - design
  - chiplet
  - bitwise
image: https://wiki.polygon.technology/img/thumbnail/polygon-miden.png
---

In this note we describe how to compute bitwise AND and XOR operations on 32-bit values and the constraints required for proving correct execution.

Assume that a and b are field elements in a 64-bit prime field. Assume also that a and b are known to contain values smaller than 2^{32}. We want to compute $a \oplus b \rightarrow z$, where $\oplus$ is either bitwise AND or XOR, and z is a field element containing the result of the corresponding bitwise operation.

First, observe that we can compute AND and XOR relations for **single bit values** as follows:

$$
and(a, b) = a \cdot b
$$

$$
xor(a, b) = a + b - 2 \cdot a \cdot b
$$

To compute bitwise operations for multi-bit values, we will decompose the values into individual bits, apply the operations to single bits, and then aggregate the bitwsie results into the final result.

To perform this operation we will use a table with 11 columns, and computing a single AND or XOR operation will require 8 table rows. We will also rely on two periodic columns as shown below.

![bitwise_execution_trace](../../assets/design/chiplets/bitwise/bitwise_execution_trace.png)

In the above, the columns have the following meanings:

- Periodic columns k_0 and k_1. These columns contain values needed to switch various constraint on or off. k_0 contains a repeating sequence of a single one, followed by seven zeros. k_1 contains a repeating sequence of seven ones, followed by a single zero.
- Input columns a and b. On the first row of each 8-row cycle, the prover will set values in these columns to the upper 4 bits of the values to which a bitwise operation is to be applied. For all subsequent rows, we will append the next-most-significant 4-bit limb to each value. Thus, by the final row columns a and b will contain the full input values for the bitwise operation.
- Columns a_0, a_1, a_2, a_3, b_0, b_1, b_2, b_3 will contain lower 4 bits of their corresponding values.
- Output column z_p. This column represents the value of column z for the prior row. For the first row, it is set to 0.
- Output column z. This column will be used to aggregate the results of bitwise operations performed over columns a_0, a_1, a_2, a_3, b_0, b_1, b_2, b_3. By the time we get to the last row in each 8-row cycle, this column will contain the final result.

Example

Let's illustrate the above table on a concrete example. For simplicity, we'll use 16-bit values, and thus, we'll only need 4 rows to complete the operation (rather than 8 for 32-bit values). Let's say $a = 41851$ (`b1010_0011_0111_1011`) and $b = 40426$ (`b1001_1101_1110_1010`), then $and(a, b) = 33130$ (`b1000_0001_0110_1010`). The table for this computation looks like so:

a	b	x0	x1	x2	x3	y0	y1	y2	y3	zp	z
10	9	0	1	0	1	1	0	0	1	0	8
163	157	1	1	0	0	1	0	1	1	8	129
2615	2526	1	1	1	0	0	1	1	1	129	2070
41851	40426	1	1	0	1	0	1	0	1	2070	33130

Here, in the first row, we set each of the a and b columns to the value of their most-significant 4-bit limb. The bit columns ($a_0 .. a_3$ and $b_0 .. b_3$) in the first row contain the lower 4 bits of their corresponding values (`b1010` and `b1001`). Column z contains the result of bitwise AND for the upper 4 bits (`b1000`), while column z_p contains that result for the prior row.

With every subsequent row, we inject the next-most-significant 4 bits of each value into the bit columns, increase the a and b columns accordingly, and aggregate the result of bitwise AND into the z column, adding it to 2^4 times the value of z in the previous row. We set column z_p to be the value of z in the prior row. By the time we get to the last row, the z column contains the result of the bitwise AND, while columns a and b contain their original values.

Constraints

AIR constraints needed to ensure the correctness of the above table are described below.

Selectors

The Bitwise chiplet supports three operations with the following operation selectors:

- `U32AND`: $s = 0$
- `U32XOR`: $s = 1$

The constraints must require that the selectors be binary and stay the same throughout the cycle:

$$
s^2 - s = 0 \text{ | degree} = 2
$$

$$
k_1 \cdot (s' - s) = 0 \text{ | degree} = 2
$$

Input decomposition

We need to make sure that inputs a and b are decomposed correctly into their individual bits. To do this, first, we need to make sure that columns a_0, a_1, a_2, a_3, b_0, b_1, b_2, b_3, can contain only binary values (0 or 1). This can be accomplished with the following constraints (for i ranging between 0 and 3):

$$
a_i^2 - a_i = 0 \text{ | degree} = 2
$$

$$
b_i^2 - b_i = 0 \text{ | degree} = 2
$$

Then, we need to make sure that on the first row of every 8-row cycle, the values in the columns a and b are exactly equal to the aggregation of binary values contained in the individual bit columns a_i, and b_i. This can be enforced with the following constraints:

$$
k_0 \cdot \left(a - \sum_{i=0}^3(2^i \cdot a_i)\right) = 0 \text{ | degree} = 2
$$

$$
k_0 \cdot \left(b - \sum_{i=0}^3(2^i \cdot b_i)\right) = 0 \text{ | degree} = 2
$$

The above constraints enforce that when $k_0 = 1$, $a = \sum_{i=0}^3(2^i \cdot a_i)$ and $b = \sum_{i=0}^3(2^i \cdot b_i)$.

Lastly, we need to make sure that for all rows in an 8-row cycle except for the last one, the values in a and b columns are increased by the values contained in the individual bit columns a_i and b_i. Denoting a as the value of column a in the current row, and a' as the value of column a in the next row, we can enforce these conditions as follows:

$$
k_1 \cdot \left(a' - \left(a \cdot 16 + \sum_{i=0}^3(2^i \cdot a'_i)\right)\right) = 0 \text{ | degree} = 2
$$

$$
k_1 \cdot \left(b' - \left(b \cdot 16 + \sum_{i=0}^3(2^i \cdot b'_i)\right)\right) = 0 \text{ | degree} = 2
$$

The above constraints enforce that when $k_1 = 1$, $a' = 16 \cdot a + \sum_{i=0}^3(2^i \cdot a'_i)$ and $b' = 16 \cdot b + \sum_{i=0}^3(2^i \cdot b'_i)$.

Output aggregation

To ensure correct aggregation of operations over individual bits, first we need to ensure that in the first row, the aggregated output value of the previous row should be 0.
$$
k_0 \cdot z_p = 0 \text{ | degree} = 2
$$

Next, we need to ensure that for each row except the last, the aggregated output value must equal the previous aggregated output value in the next row.
$$
k_1 \cdot \left(z - z'_p\right) = 0 \text{ | degree} = 2
$$

Lastly, we need to ensure that for all rows the value in the z column is computed by multiplying the previous output value (from the z_p column in the current row) by 16 and then adding it to the bitwise operation applied to the row's set of bits of a and b. The entire constraint must also be multiplied by the operation selector flag to ensure it is only applied for the appropriate operation.

For `U32AND`, this is enforced with the following constraint:

$$
(1 - s) \cdot \left(z -(z_p \cdot 16 + \sum_{i=0}^3(2^i \cdot a_i \cdot b_i))\right) = 0 \text{ | degree} = 3
$$

For `U32XOR`, this is enforced with the following constraint:

$$
s \cdot \left(z -(z_p \cdot 16 + \sum_{i=0}^3(2^i \cdot (a_i + b_i - 2 \cdot a_i \cdot b_i)))\right) = 0 \text{ | degree} = 3
$$

Bitwise chiplet bus constraints

To simplify the notation for describing bitwise constraints on the chiplet bus, we'll first define variable u, which represents how a, b, and z in the execution trace are reduced to a single value. Denoting the random values received from the verifier as α_0, α_1, etc., this can be achieved as follows.

$$
u = \alpha_1 \cdot a + \alpha_2 \cdot b + \alpha_3 \cdot z
$$

To request a bitwise operation, the prover will provide the values of a, b, and z non-deterministically to the [stack](../stack/u32_ops.md#u32and) (the component that makes bitwise requests). The lookup can then be performed by dividing $\left(\alpha_0 + u\right)$ out of the bus column:

$$
b'_{chip} \cdot \left(\alpha_0 + u\right) = b_{chip}
$$

To provide the results of bitwise operations to the chiplets bus, we want to include values of a, b and z at the last row of the cycle.

First, we'll define another intermediate variable v_i. It will include u into the product when $k_1 = 0$. (u_i represents the value of u for row i of the trace.)

$$
v_i = (1-k_1) \cdot u_i
$$

Then, setting $m = 1 - k_1$, we can compute the permutation product from the bitwise chiplet as follows:

$$
\prod_{i=0}^n ((\alpha_0 + v_i) \cdot m_i + 1 - m_i)
$$

The above ensures that when $1 - k_1 = 0$ (which is true for all rows in the 8-row cycle except for the last one), the product does not change. Otherwise, $(\alpha_0 + v_i)$ gets included into the product.

The constraints for the two sides of the bus communication are combined as follows:

$$

```
b'_{chip} \cdot \left(\alpha_0 + u_i\right) = b_{chip} \cdot ((\alpha_0 + v_i) \cdot m_i + 1 - m_i) \text{ | degree} = 4
$$
---
id: hasher
title: Hash Chiplet
sidebar_label: Hash Chiplet
description: "Hash Chiplet description."
keywords:
  - docs
  - matic
  - polygon
  - wiki
  - miden
  - design
  - chiplet
  - hash
image: https://wiki.polygon.technology/img/thumbnail/polygon-miden.png
---

Miden VM "offloads" all hash-related computations to a separate _hash processor_. This chiplet supports executing the [Rescue Prime Optimized](https://eprint.iacr.org/2022/1577) hash function (or rather a [specific instantiation](https://docs.rs/miden-crypto/latest/miden_crypto/hash/rpo/struct.Rpo256.html) of it) in the following settings:

- A single permutation of Rescue Prime Optimized.
- A simple 2-to-1 hash.
- A linear hash of $n$ field elements.
- Merkle path verification.
- Merkle root update.

The chiplet can be thought of as having a small instruction set of $11$ instructions. These instructions are listed below, and examples of how these instructions are used by the chiplet are described in the following sections.

| Instruction | Description |
| ----------- | ----------- |
| `HR`       | Executes a single round of the VM's native hash function. All cycles which are not one less than a multiple of $8$ execute this instruction. That is, the chiplet executes this instruction on cycles $0, 1, 2, 3, 4, 5, 6$, but not $7$, and then again, $8, 9, 10, 11, 12, 13, 14$, but not $15$ etc.                                |
| `BP`        | Initiates computation of a single permutation, a 2-to-1 hash, or a linear hash of many elements. This instruction can be executed only on cycles which are multiples of $8$, and it can also be executed concurrently with an `HR` instruction.                                                                |
| `MP`        | Initiates Merkle path verification computation. This instruction can be executed only on cycles which are multiples of $8$, and it can also be executed concurrently with an `HR` instruction.
|
| `MV`        | Initiates Merkle path verification for the "old" node value during Merkle root update computation. This instruction can be executed only on cycles which are multiples of $8$, and it can also be executed concurrently with an `HR` instruction.                                                                |
| `MU`        | Initiates Merkle path verification for the "new" node value during Merkle root update computation. This instruction can be executed only on cycles which are multiples of $8$, and it can also be executed concurrently with an `HR` instruction.                                                                |
| `HOUT`      | Returns the result of the currently running computation. This instruction can be executed only on cycles which are one less than a multiple of $8$ (e.g. $7$, $15$ etc.).
|
| `SOUT`      | Returns the whole hasher state. This instruction can be executed only on cycles which are one less than a multiple of $8$, and only if the computation was started using `BP` instruction.
|
| `ABP`       | Absorbs a new set of elements into the hasher state when computing a linear hash of many elements. This instruction can be executed only on cycles which are one less than a multiple of $8$, and only if the computation was started using `BP` instruction.                                                |
| `MPA`       | Absorbs the next Merkle path node into the hasher state during Merkle path verification computation. This instruction can be executed only on cycles which are one less than a multiple of $8$, and only if the computation was started using `MP` instruction.                                                |
| `MVA`       | Absorbs the next Merkle path node into the hasher state during Merkle path verification for the "old" node value during Merkle root update computation. This instruction can be executed only on cycles which are one less than a multiple of $8$, and only if the computation was started using `MV` instruction. |
| `MUA`       | Absorbs the next Merkle path node into the hasher state during Merkle path verification for the "new" node value during Merkle root update computation. This instruction can be executed only on cycles which are one less than a multiple of $8$, and only if the computation was started using `Mu` instruction. |

## Chiplet trace

Execution trace table of the chiplet consists of $17$ trace columns and $3$ periodic columns. The structure of the table is such that a single permutation of the hash function can be computed using $8$ table rows. The layout of the table is illustrated below.

![hash_execution_trace](../../assets/design/chiplets/hasher/hash_execution_trace.png)

The meaning of the columns is as follows:
```

- Three periodic columns k_0, k_1, and k_2 are used to help select the instruction executed at a given row. All of these columns contain patterns which repeat every 8 rows. For k_0 the pattern is 7 zeros followed by 1 one, helping us identify the last row in the cycle. For k_1 the pattern is 6 zeros, 1 one, and 1 zero, which can be used to identify the second-to-last row in a cycle. For k_2 the pattern is 1 one followed by 7 zeros, which can identify the first row in the cycle.
- Three selector columns s_0, s_1, and s_2. These columns can contain only binary values (ones or zeros), and they are also used to help select the instruction to execute at a given row.
- One row address column r. This column starts out at 1 and gets incremented by 1 with every row.
- Twelve hasher state columns $h_0, ..., h_{11}$. These columns are used to hold the hasher state for each round of the hash function permutation. The state is laid out as follows:
  - The first four columns ($h_0, ..., h_3$) are reserved for capacity elements of the state. When the state is initialized for hash computations, h_0 should be set to 0 if the number of elements to be hashed is a multiple of the rate width (8). Otherwise, h_0 should be set to 1. h_1 should be set to the domain value if a domain has been provided (as in the case of [control block hashing](../programs.md#program-hash-computation)). All other capacity elements should be set to 0's.
  - The next eight columns ($h_4, ..., h_{11}$) are reserved for the rate elements of the state. These are used to absorb the values to be hashed. Once the permutation is complete, hash output is located in the first four rate columns ($h_4, ..., h_7$).
- One index column i. This column is used to help with Merkle path verification and Merkle root update computations.

In addition to the columns described above, the chiplet relies on two running product columns which are used to facilitate multiset checks (similar to the ones described [here](https://hackmd.io/@arielg/ByFgSDA7D)). These columns are:

- p_0 - which is used to tie the chiplet table with the main VM's stack. That is, values representing inputs consumed by the chiplet and outputs produced by the chiplet are multiplied into p_0, while the main VM stack divides them out of p_0. Thus, if the sets of inputs and outputs between the main VM stack and hash chiplet are the same, value of p_0 should be equal to 1 at the start and the end of the execution trace.
- p_1 - which is used to keep track of the *sibling* table used for Merkle root update computations. Specifically, when a root for the old leaf value is computed, we add an entry for all sibling nodes to the table (i.e., we multiply p_1 by the values representing these entries). When the root for the new leaf value is computed, we remove the entries for the nodes from the table (i.e., we divide p_1 by the value representing these entries). Thus, if both computations used the same set of sibling nodes (in the same order), the sibling table should be empty by the time Merkle root update procedure completes (i.e., the value of p_1 would be 1).

Instruction flags

As mentioned above, chiplet instructions are encoded using a combination of periodic and selector columns. These columns can be used to compute a binary flag for each instruction. Thus, when a flag for a given instruction is set to 1, the chiplet executes this instruction. Formulas for computing instruction flags are listed below.

Flag	Value	Notes
f_{rpr}	$1 - k_0$	Set to 1 on the first 7 steps of every 8-step cycle.
f_{bp}	$k_2 \cdot s_0 \cdot (1 - s_1) \cdot (1 - s_2)$	Set to 1 when selector flags are $(1, 0, 0)$ on rows which are multiples of 8.
f_{mp}	$k_2 \cdot s_0 \cdot (1 - s_1) \cdot s_2$	Set to 1 when selector flags are $(1, 0, 1)$ on rows which are multiples of 8.
f_{mv}	$k_2 \cdot s_0 \cdot s_1 \cdot (1 - s_2)$	Set to 1 when selector flags are $(1, 1, 0)$ on rows which are multiples of 8.
f_{mu}	$k_2 \cdot s_0 \cdot s_1 \cdot s_2$	Set to 1 when selector flags are $(1, 1, 1)$ on rows which are multiples of 8.
f_{hout}	$k_0 \cdot (1 - s_0) \cdot (1 - s_1) \cdot (1 - s_2)$	Set to 1 when selector flags are $(0, 0, 0)$ on rows which are 1 less than a multiple of 8.
f_{sout}	$k_0 \cdot (1 - s_0) \cdot (1 - s_1) \cdot s_2$	Set to 1 when selector flags are $(0, 0, 1)$ on rows which are 1 less than a multiple of 8.
f_{abp}	$k_0 \cdot s_0 \cdot (1 - s_1) \cdot (1 - s_2)$	Set to 1 when selector flags are $(1, 0, 0)$ on rows which are 1 less than a multiple of 8.
f_{mpa}	$k_0 \cdot s_0 \cdot (1 - s_1) \cdot s_2$	Set to 1 when selector flags are $(1, 0, 1)$ on rows which are 1 less than a multiple of 8.
f_{mva}	$k_0 \cdot s_0 \cdot s_1 \cdot (1 - s_2)$	Set to 1 when selector flags are $(1, 1, 0)$ on rows which are 1 less than a multiple of 8.
f_{mua}	$k_0 \cdot s_0 \cdot s_1 \cdot s_2$	Set to 1 when selector flags are $(1, 1, 1)$ on rows which are 1 less than a multiple of 8.

A few additional notes about flag values:

- With the exception of f_{rpr}, all flags are mutually exclusive. That is, if one flag is set to 1, all other flats are set to 0.
- With the exception of f_{rpr}, computing flag values involves 3 multiplications, and thus the degree of these flags is 4.
- We can also define a flag $f_{out} = k_0 \cdot (1 - s_0) \cdot (1 - s_1)$. This flag will be set to 1 when either $f_{hout}=1$ or $f_{sout}=1$ in the current row.
- We can define a flag $f_{out}' = k_1 \cdot (1 - s_0') \cdot (1 - s_1')$. This flag will be set to 1 when either $f_{hout}=1$ or $f_{sout}=1$ in the next row.

We also impose the following restrictions on how values in selector columns can be updated:

- Values in columns s_1 and s_2 must be copied over from one row to the next, unless $f_{out} = 1$ or $f_{out}' = 1$ indicating the `hout` or `sout` flag is set for the current or the next row.

```
- Value in $s_0$ must be set to $1$ if $f_{out}=1$ for the previous row, and to $0$ if any of the flags $f_{abp}$, $f_{mpa}$,
$f_{mva}$, or $f_{mua}$ are set to $1$ for the previous row.

The above rules ensure that we must finish one computation before starting another, and we can't change the type of the
computation before the computation is finished.

## Computation examples

### Single permutation

Computing a single permutation of Rescue Prime Optimized hash function involves the following steps:

1. Initialize hasher state with $12$ field elements.
2. Apply Rescue Prime Optimized permutation.
3. Return the entire hasher state as output.

The chiplet accomplishes the above by executing the following instructions:
```

[BP, HR] // init state and execute a hash round (concurrently) HR HR HR HR HR HR // execute 6 more hash rounds SOUT // return the entire state as output

```
Execution trace for this computation would look as illustrated below.

![hash_1_permutation_trace](../../assets/design/chiplets/hasher/hash_1_permutation_trace.png)

In the above $\{a_0, ..., a_{11}\}$ is the input state of the hasher, and $\{b_0, ..., b_{11}\}$ is the output state of the
hasher.

### Simple 2-to-1 hash

Computing a 2-to-1 hash involves the following steps:

1. Initialize hasher state with $8$ field elements, setting the second capacity element to $domain$ if the domain is provided (as
in the case of [control block hashing](../programs.md#program-hash-computation)) or else $0$, and the remaining capacity elements
to $0$.
2. Apply Rescue Prime Optimized permutation.
3. Return elements ${4, ..., 7}$ of the hasher state as output.

The chiplet accomplishes the above by executing the following instructions:
```

[BP, HR] // init state and execute a hash round (concurrently) HR HR HR HR HR HR // execute 6 more hash rounds HOUT // return elements 4, 5, 6, 7 of the state as output

```
Execution trace for this computation would look as illustrated below.

![hash_2_to_1_hash](../../assets/design/chiplets/hasher/hash_2_to_1_hash.png)

In the above, we compute the following:

$$
\{c_0, c_1, c_2, c_3\} \leftarrow hash(\{a_0, a_1, a_2, a_3\}, \{b_0, b_1, b_2, b_3\})
$$

### Linear hash of n elements

Computing a linear hash of $n$ elements consists of the following steps:

1. Initialize hasher state with the first $8$ elements, setting the first capacity register to $0$ if $n$ is a multiple of the
rate width ($8$) or else $1$, and the remaining capacity elements to $0$.
2. Apply Rescue Prime Optimized permutation.
3. Absorb the next set of elements into the state (up to $8$ elements), while keeping capacity elements unchanged.
4. Repeat steps 2 and 3 until all $n$ elements have been absorbed.
5. Return elements ${4, ..., 7}$ of the hasher state as output.

The chiplet accomplishes the above by executing the following instructions (for hashing $16$ elements):
```

[BP, HR] // init state and execute a hash round (concurrently) HR HR HR HR HR HR // execute 6 more hash rounds ABP // absorb the next set of elements into the state HR HR HR HR HR HR HR // execute 7 hash rounds HOUT // return elements 4, 5, 6, 7 of the state as output

```
Execution trace for this computation would look as illustrated below.

![hash_linear_hash_n](../../assets/design/chiplets/hasher/hash_linear_hash_n.png)

In the above, the value absorbed into hasher state between rows $7$ and $8$ is the delta between values $t_i$ and $s_i$. Thus, if
we define $b_i = t_i - s_i$ for $i \in \{0, ..., 7\}$, the above computes the following:
```

```
$$
\{r_0, r_1, r_2, r_3\} \leftarrow hash(a_0, ..., a_7, b_0, ..., b_7)
$$

### Verify Merkle path

Verifying a Merkle path involves the following steps:

1. Initialize hasher state with the leaf and the first node of the path, setting all capacity elements to $0$s.
   a. Also, initialize the index register to the leaf's index value.
2. Apply Rescue Prime Optimized permutation.
   a. Make sure the index value doesn't change during this step.
3. Copy the result of the hash to the next row, and absorb the next node of the Merkle path into the hasher state.
   a. Remove a single bit from the index, and use it to determine how to place the copied result and absorbed node in the state.
4. Repeat steps 2 and 3 until all nodes of the Merkle path have been absorbed.
5. Return elements ${4, ..., 7}$ of the hasher state as output.
   a. Also, make sure the index value has been reduced to $0$.

The chiplet accomplishes the above by executing the following instructions (for Merkle tree of depth $3$):
```

[MP, HR] // init state and execute a hash round (concurrently) HR HR HR HR HR HR // execute 6 more hash rounds MPA // copy result & absorb the next node into the state HR HR HR HR HR HR HR // execute 7 hash rounds HOUT // return elements 4, 5, 6, 7 of the state as output

```
Suppose we have a Merkle tree as illustrated below. This Merkle tree has $4$ leaves, each of which consists of $4$ field elements.
For example, leaf $a$ consists of elements $a_0, a_1, a_2, a_3$, leaf be consists of elements $b_0, b_1, b_2, b_3$ etc.

![hash_merkle_tree](../../assets/design/chiplets/hasher/hash_merkle_tree.png)

If we wanted to verify that leaf $d$ is in fact in the tree, we'd need to compute the following hashes:

$$
r \leftarrow hash(e, hash(c, d))
$$

And if $r = g$, we can be convinced that $d$ is in fact in the tree at position $3$. Execution trace for this computation would
look as illustrated below.

![hash_merkle_tree_trace](../../assets/design/chiplets/hasher/hash_merkle_tree_trace.png)

In the above, the prover provides values for nodes $c$ and $e$ non-deterministically.

### Update Merkle root

Updating a node in a Merkle tree (which also updates the root of the tree) can be simulated by verifying two Merkle paths: the
path that starts with the old leaf and the path that starts with the new leaf.

Suppose we have the same Merkle tree as in the previous example, and we want to replace node $d$ with node $d'$. The computations
we'd need to perform are:

$$
r \leftarrow hash(e, hash(c, d)) \\
r' \leftarrow hash(e, hash(c, d'))
$$

Then, as long as $r = g$, and the same values were used for $c$ and $e$ in both computations, we can be convinced that the new
root of the tree is $r'$.

The chiplet accomplishes the above by executing the following instructions:
```

// verify the old merkle path [MV, HR] // init state and execute a hash round (concurrently) HR HR HR HR HR HR // execute 6 more hash rounds MPV // copy result & absorb the next node into the state HR HR HR HR HR HR HR // execute 7 hash rounds HOUT // return elements 4, 5, 6, 7 of the state as output

// verify the new merkle path [MU, HR] // init state and execute a hash round (concurrently) HR HR HR HR HR HR // execute 6 more hash rounds MPU // copy result & absorb the next node into the state HR HR HR HR HR HR HR // execute 7 hash rounds HOUT // return elements 4, 5, 6, 7 of the state as output

```
The semantics of `MV` and `MU` instructions are similar to the semantics of `MP` instruction from the previous example (and `MVA`
and `MUA` are similar to `MPA`) with one important difference: `MV*` instructions add the absorbed node (together with its index
in the tree) to permutation column $p_1$, while `MU*` instructions remove the absorbed node (together with its index in the tree)
from $p_1$. Thus, if the same nodes were used during both Merkle path verification, the state of $p_1$ should not change. This
mechanism is used to ensure that the same internal nodes were used in both computations.

## AIR constraints

When describing AIR constraints, we adopt the following notation: for column $x$, we denote the value in the current row simply as
$x$, and the value in the next row of the column as $x'$. Thus, all transition constraints described in this note work with two
```

consecutive rows of the execution trace.

Row address constraint

As mentioned above, row address r starts at 1, and is incremented by 1 with every row. The first condition can be enforced with a boundary constraint which specifies $r=1$ at the first row. The second condition can be enforced via the following transition constraint:

$$
r' - r - 1 = 0 \text{ | degree} = 1
$$

This constraint should not be applied to the very last row of the hasher execution trace, since we do not want to enforce a value that would conflict with the first row of a subsequent chiplet in the Chiplets module. Therefore we can create a special virtual flag for this constraint using the $chip_s_0$ selector column from the [Chiplets](main.md) module that selects for the hash chiplet.

The modified row address constraint which should be applied is the following:

$$
(1 - chip_s_0') \cdot (r' - r - 1) = 0 \text{ | degree} = 2
$$

Note: this constraint should also be multiplied by Chiplets module's selector flag s_0, as is true for all constraints in this chiplet.

Selector columns constraints

For selector columns, first we must ensure that only binary values are allowed in these columns. This can be done with the following constraints:

$$
s_0^2 - s_0 = 0 \text{ | degree} = 2 \\
s_1^2 - s_1 = 0 \text{ | degree} = 2 \\
s_2^2 - s_2 = 0 \text{ | degree} = 2
$$

Next, we need to make sure that unless $f_{out}=1$ or $f_{out}'=1$, the values in columns s_1 and s_2 are copied over to the next row. This can be done with the following constraints:

$$
(s_1' - s_1) \cdot (1 - f_{out}') \cdot (1 - f_{out}) = 0 \text{ | degree} = 9 \\
(s_2' - s_2) \cdot (1 - f_{out}') \cdot (1 - f_{out}) = 0 \text{ | degree} = 9
$$

Next, we need to enforce that if any of $f_{abp}, f_{mpa}, f_{mva}, f_{mua}$ flags is set to 1, the next value of s_0 is 0. In all other cases, s_0 should be unconstrained. These flags will only be set for rows that are 1 less than a multiple of 8 (the last row of each cycle). This can be done with the following constraint:

$$
s_0' \cdot (f_{abp} + f_{mpa} + f_{mva} + f_{mua})= 0 \text{ | degree} = 5
$$

Lastly, we need to make sure that no invalid combinations of flags are allowed. This can be done with the following constraints:

$$
k_0 \cdot (1 - s_0) \cdot s_1 = 0 \text{ | degree} = 3
$$

The above constraints enforce that on every step which is one less than a multiple of 8, if $s_0 = 0$, then s_1 must also be set to 0. Basically, if we set $s_0=0$, then we must make sure that either $f_{hout}=1$ or $f_{sout}=1$.

Node index constraints

Node index column i is relevant only for Merkle path verification and Merkle root update computations, but to simplify the overall constraint system, the same constraints will be imposed on this column for all computations.

Overall, we want values in the index column to behave as follows:

- When we start a new computation, we should be able to set i to an arbitrary value.
- When a computation is finished, value in i must be 0.
- When we absorb a new node into the hasher state we must shift the value in i by one bit to the right.
- In all other cases value in i should not change.

A shift by one bit to the right can be described with the following equation: $i = 2 \cdot i' + b$, where b is the value of the bit which is discarded. Thus, as long as b is a binary value, the shift to the right is performed correctly, and this can be enforced with the following constraint:

$$
b^2 - b = 0
$$

Since we want to enforce this constraint only when a new node is absorbed into the hasher state, we'll define a flag for when this should happen as follows:

$$
f_{an} = f_{mp} + f_{mv} + f_{mu} + f_{mpa} + f_{mva} + f_{mua}
$$

And then the full constraint would looks as follows:

$$
f_{an} \cdot (b^2 - b) = 0 \text{ | degree} = 6
$$

Next, to make sure when a computation is finished $i=0$, we can use the following constraint:

$$
f_{out} \cdot i = 0 \text{ | degree} = 5
$$

Finally, to make sure that the value in i is copied over to the next row unless we are absorbing a new row or the computation is finished, we impose the following constraint:

$$
(1 - f_{an} - f_{out}) \cdot (i' - i) = 0 \text{ | degree} = 5
$$

To satisfy these constraints for computations not related to Merkle paths (i.e., 2-to-1 hash and liner hash of elements), we can set $i = 0$ at the start of the computation. This guarantees that i will remain 0 until the end of the computation.

Hasher state constraints

Hasher state columns $h_0, ..., h_{11}$ should behave as follows:

- For the first 7 row of every 8-row cycle (i.e., when $k_0=0$), we need to apply [Rescue Prime Optimized](https://eprint.iacr.org/2022/1577) round constraints to the hasher state. For brevity, we omit these constraints from this note.
- On the 8th row of every 8-row cycle, we apply the constraints based on which transition flag is set as described in the table below.

Specifically, when absorbing the next set of elements into the state during linear hash computation (i.e., $f_{abp} = 1$), the first 4 elements (the capacity portion) are carried over to the next row. For $j \in \{0, ..., 3\}$ this can be described as follows:

$$
f_{abp} \cdot (h'_j - h_j) = 0 \text{ | degree} = 5
$$

When absorbing the next node during Merkle path computation (i.e., $f_{mp} + f_{mv} + f_{mu}=1$), the result of the previous hash ($h_4, ..., h_7$) are copied over either to $(h_4', ..., h_7')$ or to $(h_8', ..., h_{11}')$ depending on the value of b, which is defined in the same way as in the previous section. For $j \in \{0, ..., 3\}$ this can be described as follows:

$$
(f_{mp} + f_{mv} + f_{mu}) \cdot ((1 - b) \cdot (h_{j +4}' - h_{j+4}) + b \cdot (h_{j + 8}' - h_{j + 4})) = 0 \text{ | degree} = 6
$$

Note, that when a computation is completed (i.e., $f_{out}=1$), the next hasher state is unconstrained.

Multiset check constraints
In this sections we describe constraints which enforce updates for [multiset check columns](../multiset.md) p_0 and p_1. These columns can be updated only on rows which are multiples of 8 or 1 less than a multiple of 8. On all other rows the values in the columns remain the same.

To simplify description of the constraints, we define the following variables. Below, we denote random values sent by the verifier after the prover commits to the main execution trace as α_0, α_1, α_2 etc..

$$
m = op_{label} + 2^4 \cdot k_0 + 2^5 \cdot k_2 \\
v_h = \alpha_0 + \alpha_1 \cdot m + \alpha_2 \cdot r + \alpha_3 \cdot i \\
v_a = \sum_{j=0}^{3}(\alpha_{j+4} \cdot h_j) \\
v_b = \sum_{j=4}^{7}(\alpha_{j+4} \cdot h_j) \\
v_c = \sum_{j=8}^{11}(\alpha_{j+4} \cdot h_j) \\
v_d = \sum_{j=8}^{11}(\alpha_j \cdot h_j) \\
$$

In the above:

- m is a _transition label_, composed of the [operation label](main.md#operation-labels) and the periodic columns that uniquely identify each transition function. The values in the k_0 and k_2 periodic columns are included to identify the row in the hash cycle where the operation occurs. They serve to differentiate between operations that share selectors but occur at different rows in the cycle, such as `BP`, which uses $op_{linhash}$ at the first row in the cycle to initiate a linear hash, and `ABP`, which uses $op_{linhash}$ at the last row in the cycle to absorb new elements.
- v_h is a _common header_ which is a combination of transition label, row address, and node index.
- v_a, v_b, v_c are the first, second, and third words (4 elements) of the hasher state.
- v_d is the third word of the hasher state but computed using the same α values as used for the second word. This is needed for computing the value of v_{leaf} below to ensure that the same α values are used for the leaf node regardless of which part of the state the node comes from.

Hash chiplet bus constraints
As described previously, running product column p_0 is used to tie the hash chiplet with the main VM's stack. When receiving inputs from or returning results to the stack, hash chiplet multiplies p_0 by their respective values. On other other side, when sending inputs to the hash chiplet or receiving results from the chiplet, the stack divides p_0 by their values.

In the section below we describe only the hash chiplet side of the constraints (i.e., multiplying p_0 by relevant values). We define values which are to be multiplied into the p_0 for each operation as follows:

When starting a new simple or linear hash computation (i.e., $f_{bp}=1$) or when returning the entire state of the hasher ($f_{sout}=1$), the entire hasher state is included into p_0:
$$
v_{all} = v_h + v_a + v_b + v_c
$$

When starting a Merkle path computation (i.e., $f_{mp} + f_{mv} + f_{mu} = 1$), we include the leaf of the path into p_0. The leaf is selected from the state based on value of b (defined as in the previous section):
$$
v_{leaf} = v_h + (1-b) \cdot v_b + b \cdot v_d
$$

When absorbing a new set of elements into the state while computing a linear hash (i.e., $f_{abp}=1$), we include deltas between the last 8 elements of the hasher state (the rate) into p_0:
$$
v_{abp} = v_h + v'_b + v'_c - (v_b + v_c)
$$

When a computation is complete (i.e., $f_{hout}=1$), we include the second word of the hasher state (the result) into p_0:
$$
v_{res} = v_h + v_b
$$

Using the above values, we can describe the constraints for updating column p_0 as follows.

$$
p_0' = p_0 \cdot ((f_{bp} + f_{sout}) \cdot v_{all} + (f_{mp} + f_{mv} + f_{mu}) \cdot v_{leaf} + f_{abp} \cdot v_{abp} + f_{hout} \cdot v_{res} + \\
1 - (f_{bp} + f_{mp} + f_{mv} + f_{mu} + f_{abp} + f_{out}))
$$

The above constraint reduces to the following under various flag conditions:

Condition	Applied constraint
$f_{bp} = 1$	$p_0' = p_0 \cdot v_{all}$
$f_{sout} = 1$	$p_0' = p_0 \cdot v_{all}$
$f_{mp} = 1$	$p_0' = p_0 \cdot v_{leaf}$
$f_{mv} = 1$	$p_0' = p_0 \cdot v_{leaf}$
$f_{mu} = 1$	$p_0' = p_0 \cdot v_{leaf}$
$f_{abp} = 1$	$p_0' = p_0 \cdot v_{abp}$
$f_{hout} = 1$	$p_0' = p_0 \cdot v_{res}$
Otherwise	$p_0' = p_0$

Note that the degree of the above constraint is 7.

Sibling table constraints
As mentioned previously, sibling table (represented by running column p_1) is used to keep track of sibling nodes used during Merkle root update computation. For this computation, we need to enforce the following rules:
* When computing the old Merkle root, whenever a new sibling node is absorbed into the hasher state (i.e., $f_{mv} + f_{mva} = 1$), an entry for this sibling should be included into p_1.
* When computing the new Merkle root, whenever a new sibling node is absorbed into the hasher state (i.e., $f_{mu} + f_{mua} = 1$), the entry for this sibling should be removed from p_1.

To simplify the description of the constraints, we use variables v_b and v_c defined above and define the value representing an entry in the sibling table as follows:
$$
v_{sibling} = \alpha_0 + \alpha_3 \cdot i + b \cdot v_b + (1-b) \cdot v_c

$$

Using the above value, we can define the constraint for updating p_1 as follows:

$$
p_1' \cdot \left((f_{mv} + f_{mva}) \cdot v_{sibling} + 1 - (f_{mv} + f_{mva}) \right) = \\
p_1 \cdot \left((f_{mu} + f_{mua}) \cdot v_{sibling} + 1 - (f_{mu} + f_{mua}) \right)
$$

The above constraint reduces to the following under various flag conditions:

Condition	Applied constraint
$f_{mv} = 1$	$p_1' \cdot v_{sibling} = p_1$
$f_{mva} = 1$	$p_1' \cdot v_{sibling} = p_1$
$f_{mu} = 1$	$p_1' = p_1 \cdot v_{sibling}$
$f_{mua} = 1$	$p_1' = p_1 \cdot v_{sibling}$
Otherwise	$p_1' = p_1$

Note that the degree of the above constraint is 7.

To make sure computation of the old Merkle root is immediately followed by the computation of the new Merkle root, we impose the following constraint:

$$
(f_{bp} + f_{mp} + f_{mv}) \cdot (1 - p_1) = 0 \text{ | degree} = 5
$$

The above means that whenever we start a new computation which is not the computation of the new Merkle root, the sibling table must be empty. Thus, after the hash chiplet computes the old Merkle root, the only way to clear the table is to compute the new Merkle root.

Together with boundary constraints enforcing that $p_1=1$ at the first and last rows, the above constraints ensure that if a node was included into p_1 as a part of computing the old Merkle root, the same node must be removed from p_1 as a part of computing the new Merkle root.

---
id: main
title: Chiplets
sidebar_label: Chiplets
description: "Chiplets module components."
keywords:
  - docs
  - matic
  - polygon
  - wiki
  - miden
  - design
  - chiplet
  - hash
  - bitwise
  - memory
image: https://wiki.polygon.technology/img/thumbnail/polygon-miden.png
---

The Chiplets module contains specialized components dedicated to accelerating complex computations. Each chiplet specializes in executing a specific type of computation and is responsible for proving both the correctness of its computations and its own internal consistency.

Currently, Miden VM relies on 3 chiplets:

- The [Hash Chiplet](./hasher.md) (also referred to as the Hasher), used to compute Rescue Prime Optimized hashes both for sequential hashing and for Merkle tree hashing.
- The [Bitwise Chiplet](./bitwise.md), used to compute bitwise operations (e.g., `AND`, `XOR`) over 32-bit integers.
- The [Memory Chiplet](./memory.md), used to support random-access memory in the VM.

Each chiplet executes its computations separately from the rest of the VM and proves the internal correctness of its execution trace in a unique way that is specific to the operation(s) it supports. These methods are described by each chiplet’s documentation.

Chiplet bus

The chiplets must be explicitly connected to the rest of the VM in order for it to use their operations. This connection must prove that all specialized operations which a given VM component claimed to offload to one of the chiplets were in fact executed by the correct chiplet with the same set of inputs and outputs as those used by the offloading component.

This is achieved via a [bus](../multiset.md#communication-buses) called b_{chip} where a request can be sent to any chiplet and a corresponding response will be sent back by that chiplet.

The bus is implemented as a single [running product column](../multiset.md) where:

- Each request is “sent” by computing an operation-specific lookup value from an [operation-specific label](#operation-labels), the operation inputs, and the operation outputs, and then dividing it out of the b_{chip} running product column.
- Each chiplet response is “sent” by computing the same operation-specific lookup value from the label, inputs, and outputs, and then multiplying it into the b_{chip} running product column.

Thus, if the requests and responses match, then the bus column b_{chip} will start and end with the value 1. This condition is enforced by boundary constraints on the b_{chip} column.

Note that the order of the requests and responses does not matter, as long as they are all included in b_{chip}. In fact, requests and responses for the same operation will generally occur at different cycles.

Chiplet bus constraints

The chiplet bus constraints are defined by the components that use it to communicate.

Lookup requests are sent to the chiplets bus by the following components:

- The stack sends requests for [bitwise](../stack/u32_ops.md#u32and), [memory](../stack/io_ops.md#memory-access-operations), and [cryptographic hash operations](../stack/crypto_ops.md).
- The decoder sends requests for [hash operations](../decoder/main.md#program-block-hashing) for program block hashing.

Responses are provided by the [hash](./hasher.md#hash-chiplet-bus-constraints), [bitwise](./bitwise.md#bitwise-chiplet-bus-constraints), and [memory](./memory.md#memory-row-value) chiplets.

Chiplets module trace

The execution trace of the Chiplets module is generated by stacking the execution traces of each of its chiplet components. Because each chiplet is expected to generate significantly fewer trace rows than the other VM components (i.e., the decoder, stack, and range checker), stacking them enables the same functionality without adding as many columns to the execution trace.

Each chiplet is identified within the Chiplets module by one or more chiplet selector columns which cause its constraints to be selectively applied.

The result is an execution trace of 18 trace columns, which allows space for the widest chiplet component (the Hasher) and a column to select for it.

![chiplets](../../assets/design/chiplets/chiplets.png)

During the finalization of the overall execution trace, the chiplets' traces (including internal selectors) are appended to the trace of the Chiplets module one after another, as pictured. Thus, when one chiplet's trace ends, the trace of the next chiplet starts in the subsequent row.

Additionally, a padding segment is added to the end of the Chiplets module's trace so that the number of rows in the table always matches the overall trace length of the other VM processors, regardless of the length of the chiplet traces. The padding will simply contain zeroes.

Chiplets order

The order in which the chiplets are stacked is determined by the requirements of each chiplet, including the width of its execution trace and the degree of its constraints.

For simplicity, all of the "cyclic" chiplets which operate in multi-row cycles and require starting at particular row increments should come before any non-cyclic chiplets, and these should be ordered from longest-cycle to shortest-cycle. This avoids any additional alignment padding between chiplets.

After that, chiplets are ordered by degree of constraints so that higher-degree chiplets get lower-degree chiplet selector flags.

The resulting order is as follows:

Chiplet	Cycle Length	Internal Degree	Chiplet Selector Degree	Total Degree	Columns	Chiplet Selector Flag
Hash chiplet	8	8	1	9	17	$\{0\}$
Bitwise chiplet	8	3	2	5	13	$\{1, 0\}$
Memory	-	6	3	9	12	$\{1, 1, 0\}$
Padding	-	-	-	-	-	$\{1, 1, 1\}$

Additional requirements for stacking execution traces

Stacking the chiplets introduces one new complexity. Each chiplet proves its own correctness with its own set of internal transition constraints, many of which are enforced between each row in its trace and the next row. As a result, when the chiplets are stacked, transition constraints applied to the final row of one chiplet will cause a conflict with the first row of the following chiplet.

This is true for any transition constraints which are applied at every row and selected by a `Chiplet Selector Flag` for the current row. (Therefore cyclic transition constraints controlled by periodic columns do not cause an issue.)

This requires the following adjustments for each chiplet.

In the hash chiplet: the [row address constraint](hasher.md#row-address-constraint) causes a conflict that is resolved by using a virtual flag to exclude the last row. This increases the degree of the row address constraint to 2.

In the bitwise chiplet: there is no conflict, and therefore no change, since all constraints are periodic.

In the memory chiplet: all transition constraints cause a conflict. To adjust for this, the selector flag for the memory chiplet is designed to exclude its last row. Thus, memory constraints will not be applied when transitioning from the last row of the memory chiplet to the subsequent row. This is achieved without any additional increase in the degree of constraints by using s'_2 as a selector instead of s_2 as seen [below](#chiplet-constraints).

Operation labels

Each operation supported by the chiplets is given a unique identifier to ensure that the requests and responses sent to b_{chip} are indeed processed by the intended chiplet for that operation and that chiplets which support more than one operation execute the correct one.

The labels are composed from the flag values of the chiplet selector(s) and internal operation selectors (if applicable). The unique label of the operation is computed as the binary aggregation of the combined selectors plus 1, note that the combined flag is represented in big-endian, so the bit representation below is reverted.

Operation	Chiplet Selector Flag	Internal Selector Flag	Combined Flag	Label
`HASHER_LINER_HASH`	$\{0\}$	$\{1, 0, 0\}$	$\{0, 1, 0, 0\}$	3
`HASHER_MP_VERIFY`	$\{0\}$	$\{1, 0, 1\}$	$\{0, 1, 0, 1\}$	11
`HASHER_MR_UPDATE_OLD`	$\{0\}$	$\{1, 1, 0\}$	$\{0, 1, 1, 0\}$	7
`HASHER_MR_UPDATE_NEW`	$\{0\}$	$\{1, 1, 1\}$	$\{0, 1, 1, 1\}$	15
`HASHER_RETURN_HASH`	$\{0\}$	$\{0, 0, 0\}$	$\{0, 0, 0, 0\}$	1
`HASHER_RETURN_STATE`	$\{0\}$	$\{0, 0, 1\}$	$\{0, 0, 0, 1\}$	9
`BITWISE_AND`	$\{1, 0\}$	$\{0\}$	$\{1, 0, 0\}$	2
`BITWISE_XOR`	$\{1, 0\}$	$\{1\}$	$\{1, 0, 1\}$	6
`MEMORY_READ`	$\{1, 1, 0\}$	$\{1\}$	$\{1, 1, 0, 1\}$	12
`MEMORY_WRITE`	$\{1, 1, 0\}$	$\{0\}$	$\{1, 1, 0, 0\}$	4

Chiplets module constraints

Chiplet constraints

Each chiplet's internal constraints are defined in the documentation for the individual chiplets. To ensure that constraints are only ever selected for one chiplet at a time, the module's selector columns s_0, s_1, s_2 are combined into flags. Each chiplet's internal constraints are multiplied by its chiplet selector flag, and the degree of each constraint is correspondingly increased.

This gives the following sets of constraints:

$$
(1 - s_0) \cdot c_{hash} = 0 \text{ | degree} = 1 + \deg(c_{hash})
$$

$$
s_0 \cdot (1 - s_1) \cdot c_{bitwise} = 0 \text{ | degree} = 2 + \deg(c_{bitwise})
$$

$$
s_0 \cdot s_1 \cdot (1 - s'_2) \cdot c_{memory} = 0 \text{ | degree} = 3 + \deg(c_{memory})
$$

In the above:
- $c_{hash}, c_{bitwise}, c_{memory}$ each represent an internal constraint from the indicated chiplet.
- $\deg(c)$ indicates the degree of the specified constraint.
- flags are applied in a like manner for all internal constraints in each respective chiplet.
- the selector for the memory chiplet excludes the last row of the chiplet (as discussed [above](#additional-requirements-for-stacking-execution-traces)).

Chiplet selector constraints

We also need to ensure that the chiplet selector columns are set correctly. Although there are three columns for chiplet selectors, the stacked trace design means that they do not all act as selectors for the entire trace. Thus, selector constraints should only be applied to selector columns when they are acting as selectors.

- s_0 acts as a selector for the entire trace.
- s_1 acts as a selector column when $s_0 = 1$.
- s_2 acts as a selector column when $s_0 = 1$ and $s_1 = 1$.

Two conditions must be enforced for columns acting as chiplet selectors.

1. When acting as a selector, the value in the selector column must be binary.

2. When acting as a selector, the value in the selector column may only change from $0 \rightarrow 1$.

The following constraints ensure that selector values are binary.

$$
s_0^2 - s_0 = 0 \text{ | degree} = 2 \\
s_0 \cdot (s_1^2 - s_1) = 0 \text{ | degree} = 3 \\
s_0 \cdot s_1 \cdot (s_2^2 - s_2) = 0 \text{ | degree} = 4
$$

The following constraints ensure that the chiplets are stacked correctly by restricting selector values so they can only change from $0 \rightarrow 1$.

$$
s_0 \cdot (s_0 - s'_0) = 0 \text{ | degree} = 2 \\
s_0 \cdot s_1 \cdot (s_1 - s'_1) \text{ | degree} = 3 \\
s_0 \cdot s_1 \cdot s_2 \cdot (s_2 - s'_2) \text{ | degree} = 4 \\
$$

In other words, the above constraints enforce that if a selector is 0 in the current row, then it must be either 0 or 1 in the next row; if it is 1 in the current row, it must be 1 in the next row.

---
id: memory
title: Memory Chiplet
sidebar_label: Memory Chiplet
description: "Memory Chiplet description."
keywords:
  - docs
  - matic
  - polygon
  - wiki
  - miden
  - design
  - chiplet
  - memory
image: https://wiki.polygon.technology/img/thumbnail/polygon-miden.png
---

Miden VM supports linear read-write random access memory. This memory is word-addressable, meaning, four values are located at each address, and we can read and write values to/from memory in batches of four. Each value is a field element in a 64-bit prime field with modulus $2^{64} - 2^{32} + 1$. Memory address can be any field element.

In this note we describe the rational for selecting the above design and describe AIR constraints needed to support it.

The design makes extensive use of 16-bit range checks. An efficient way of implementing such range checks is described [here](../range.md).

Alternative designs

The simplest (and most efficient) alternative to the above design is contiguous write-once memory. To support such memory, we need to allocate just two trace columns as illustrated below.

![memory_alternative_design](../../assets/design/chiplets/memory/memory_alternative_design.png)

In the above, `addr` column holds memory address, and `value` column holds the field element representing the value stored at this address. Notice that some rows in this table are duplicated. This is because we need one row per memory access (either read or write operation). In the example above, value b was first stored at memory address 1, and then read from this address.

The AIR constraints for this design are very simple. First, we need to ensure that values in the `addr` column either remain the same or are incremented by 1 as we move from one row to the next. This can be achieved with the following constraint:

$$
(a' - a) \cdot (a' - a - 1) = 0
$$

where a is the value in `addr` column in the current row, and a' is the value in this column in the next row.

Second, we need to make sure that if the value in the `addr` column didn't change, the value in the `value` column also remained the same (i.e., a value stored in a given address can only be set once). This can be achieved with the following constraint:

$$
(v' - v) \cdot (a' - a - 1) = 0
$$

where v is the value in `value` column at the current row, and v' is the value in this column in the next row.

In addition to the above constraints we would also need to impose constraints needed for permutation checks, but we omit these constraints here because they are needed for all designs described in this note.

As mentioned above, this approach is very efficient: each memory access requires just 2 trace cells.

Read-write memory

Write-once memory is tricky to work with, and many developers may need to climb a steep learning curve before they become comfortable working in this model. Thus, ideally, we'd want to support read-write memory. To do this, we need to introduce additional columns as illustrated below.

![memory_read_write](../../assets/design/chiplets/memory/memory_read_write.png)

In the above, we added `clk` column, which keeps track of the clock cycle at which memory access happened. We also need to differentiate between memory reads and writes. To do this, we now use two columns to keep track of the value: `old val` contains the value stored at the address before the operation, and `new val` contains the value after the operation. Thus, if `old val` and `new val` are the same, it was a read operation. If they are different, it was a write operation.

The AIR constraints needed to support the above structure are as follows.

We still need to make sure memory addresses are contiguous:

$$
(a' - a) \cdot (a' - a - 1) = 0
$$

Whenever memory address changes, we want to make sure that `old val` is set to 0 (i.e., our memory is always initialized to 0). This can be done with the following constraint:

$$
(a' - a) \cdot v_{old}' = 0
$$

On the other hand, if memory address doesn't change, we want to make sure that `new val` in the current row is the same as `old val` in the next row. This can be done with the following constraint:

$$
(1 + a - a') \cdot (v_{new} - v_{old}') = 0
$$

Lastly, we need to make sure that for the same address values in `clk` column are always increasing. One way to do this is to perform a 16-bit range check on the value of $(i' - i - 1)$, where i is the reference to `clk` column. However, this would mean that memory operations involving the same address must happen within 65536 VM cycles from each other. This limitation would be difficult to enforce statically. To remove this limitation, we need to add two more columns as shown below:

![memory_limitation_diagram](../../assets/design/chiplets/memory/memory_limitation_diagram.png)

In the above column `d0` contains the lower 16 bits of $(i' - i - 1)$ while `d1` contains the upper 16 bits. The constraint needed to enforces this is as follows:

$$
(1 + a - a') \cdot ((i' - i - 1) - (2^{16} \cdot d_1' + d_0')) = 0
$$

Additionally, we need to apply 16-bit range checks to columns `d0` and `d1`.

Overall, the cost of reading or writing a single element is now 6 trace cells and 2 16-bit range-checks.

Non-contiguous memory

Requiring that memory addresses are contiguous may also be a difficult limitation to impose statically. To remove this limitation, we need to introduce one more column as shown below:

![memory_non_contiguous_memory](../../assets/design/chiplets/memory/memory_non_contiguous_memory.png)

In the above, the prover sets the value in the new column `t` to 0 when the address doesn't change, and to $1 / (a' - a)$ otherwise. To simplify constraint description, we'll define variable n computed as follows:

$$
n = (a' - a) \cdot t'
$$

Then, to make sure the prover sets the value of t correctly, we'll impose the following constraints:

$$
n^2 - n = 0 \\
(1 - n) \cdot (a' - a) = 0
$$

The above constraints ensure that $n=1$ whenever the address changes, and $n=0$ otherwise. We can then define the following

constraints to make sure values in columns `d0` and `d1` contain either the delta between addresses or between clock cycles.

Condition	Constraint	Comments
$n=1$	$(a' - a) - (2^{16} \cdot d_1' + d_0') = 0$	When the address changes, columns `d0` and `d1` at the next row should contain the delta between the old and the new address.
$n=0$	$(i' - i - 1) - (2^{16} \cdot d_1' + d_0') = 0$	When the address remains the same, columns `d0` and `d1` at the next row should contain the delta between the old and the new clock cycle.

We can combine the above constraints as follows:

$$
\left(n \cdot (a' - a) + (1 - n) \cdot (i' - i - 1)\right) - (2^{16} \cdot d_1' + d_0') = 0
$$

The above constraint, in combination with 16-bit range checks against columns `d0` and `d1` ensure that values in `addr` and `clk` columns always increase monotonically, and also that column `addr` may contain duplicates, while values in `clk` column must be unique for a given address.

Context separation

In many situations it may be desirable to assign memories to different context. For example, when making a cross-contract calls, memories of the caller and the callee should be separate. That is, caller should not be able to access the memory of the callee and vice-versa.

To accommodate this feature, we need to add one more column as illustrated below.

![memory_context_separation](../../assets/design/chiplets/memory/memory_context_separation.png)

This new column `ctx` should behave similarly to the address column: values in it should increase monotonically, and there could be breaks between them. We also need to change how the prover populates column `t`:

- If the context changes, `t` should be set to the inverse $(c' - c)$, where c is a reference to column `ctx`.
- If the context remains the same but the address changes, column `t` should be set to the inverse of $(a' - a)$.
- Otherwise, column `t` should be set to 0.

To simplify description of constraints, we'll define two variables n_0 and n_1 as follows:

$$
n_0 = (c' - c) \cdot t' \\
n_1 = (a' - a) \cdot t'
$$

Thus, $n_0 = 1$ when the context changes, and 0 otherwise. Also, $(1 - n_0) \cdot n_1 = 1$ when context remains the same and address changes, and 0 otherwise.

To make sure the prover sets the value of column `t` correctly, we'll need to impose the following constraints:

$$
n_0^2 - n_0 = 0 \\
(1 - n_0) \cdot (c' - c) = 0 \\
(1 - n_0) \cdot (n_1^2 - n_1) = 0 \\
(1 - n_0) \cdot (1 - n_1) \cdot (a' - a) = 0
$$

We can then define the following constraints to make sure values in columns `d0` and `d1` contain the delta between contexts, between addresses, or between clock cycles.

Condition	Constraint	Comments
$n_0=1$	$(c' - c) - (2^{16} \cdot d_1' + d_0') = 0$	When the context changes, columns `d0` and `d1` at the next row should contain the delta between the old and the new contexts.
$n_0=0$ </br> $n_1=1$	$(a' - a) - (2^{16} \cdot d_1' + d_0') = 0$	When the context remains the same but the address changes, columns `d0` and `d1` at the next row should contain the delta between the old and the new addresses.
$n_0=0$ </br> $n_1=0$	$(i' - i - 1) - (2^{16} \cdot d_1' + d_0') = 0$	When both the context and the address remain the same, columns `d0` and `d1` at the next row should contain the delta between the old and the new clock cycle.

We can combine the above constraints as follows:

$$
\left(n_0 \cdot (c' - c) + (1 - n_0) \cdot \left(n_1 \cdot (a - a') + (1 - n_1) \cdot (i' - i - 1) \right) \right) - (2^{16} \cdot d_1' + d_0') = 0
$$

The above constraint, in combination with 16-bit range checks against columns `d0` and `d1` ensure that values in `ctx`, `addr`, and `clk` columns always increase monotonically, and also that columns `ctx` and `addr` may contain duplicates, while the values in column `clk` must be unique for a given combination of `ctx` and `addr`.

Notice that the above constraint has degree 5.

Miden approach

While the approach described above works, it comes at significant cost. Reading or writing a single value requires 8 trace cells and 2 16-bit range checks. Assuming a single range check requires roughly 2 trace cells, the total number of trace cells needed grows to 12. This is about 6x worse the simple contiguous write-once memory described earlier.

Miden VM frequently needs to deal with batches of 4 field elements, which we call _words_. For example, the output of Rescue Prime Optimized hash function is a single word. A single 256-bit integer value can be stored as two words (where each element contains one 32-bit value). Thus, we can optimize for this common use case by making the memory _word-addressable_. That is 4 field elements are located at each memory address, and we can read and write elements to/from memory in batches of four.

The layout of Miden VM memory table is shown below:

![memory_miden_vm_layout](../../assets/design/chiplets/memory/memory_miden_vm_layout.png)

where:

- `s0` is a selector column which is set to 1 for read operations and 0 for write operations.
- `s1` is a selector oclumn which is set to 1 when previously accessed memory is being read and 0 otherwise. In other words, it is set to 1 only when the context and address are the same as they were in the previous row and the `s0` operation selector is set to 1 (indicating a read).
- `ctx` contains context ID. Values in this column must increase monotonically but there can be gaps between two consecutive values of up to 2^{32}. Also, two consecutive values can be the same. In AIR constraint description below, we refer to this column as c.
- `addr` contains memory address. Values in this column must increase monotonically for a given context but there can be gaps between two consecutive values of up to 2^{32}. Also, two consecutive values can be the same. In AIR constraint description below, we refer to this column as a.
- `clk` contains clock cycle at which the memory operation happened. Values in this column must increase monotonically for a given context and memory address but there can be gaps between two consecutive values of up to 2^{32}. In AIR constraint description below, we refer to this column as i.
- `v0, v1, v2, v3` columns contain field elements stored at a given context/address/clock cycle after the memory operation.
- Columns `d0` and `d1` contain lower and upper 16 bits of the delta between two consecutive context IDs, addresses, or clock cycles. Specifically:
  - When the context changes, these columns contain $(c' - c)$.
  - When the context remains the same but the address changes, these columns contain $(a' - a)$.
  - When both the context and the address remain the same, these columns contain $(i' - i - 1)$.
- Column `t` contains the inverse of the delta between two consecutive context IDs, addresses, or clock cycles. Specifically:
  - When the context changes, this column contains the inverse of $(c' - c)$.
  - When the context remains the same but the address changes, this column contains the inverse of $(a' - a)$.
  - When both the context and the address remain the same, this column contains the inverse of $(i' - i - 1)$.

For every memory access operation (i.e., read or write), a new row is added to the memory table. For read operations, `s0` is set to 1. If neither `ctx` nor `addr` have changed, then `s1` is set to 1 and the `v` columns are set to equal the values from the previous row. If `ctx` or `addr` have changed, then `s1` is set to 0 and the `v` columns are initialized to 0. For write operations, the values may be different, and both selector columns `s0` and `s1` are set to 0.

The amortized cost of reading or writing a single value is between 4 and 5 trace cells (this accounts for the trace cells needed for 16-bit range checks). Thus, from performance standpoint, this approach is roughly 2.5x worse than the simple contiguous write-once memory described earlier. However, our view is that this trade-off is worth it given that this approach provides read-write memory, context separation, and eliminates the contiguous memory requirement.

AIR constraints

To simplify description of constraints, we'll define two variables n_0 and n_1 as follows:

$$
n_0 = \Delta c \cdot t' \\
n_1 = \Delta a \cdot t'
$$

Where $\Delta c = c' - c$ and $\Delta a = a' - a$.

To make sure the prover sets the value of column `t` correctly, we'll need to impose the following constraints:

$$
n_0^2 - n_0 = 0 \text{ | degree} = 4
$$

$$
(1 - n_0) \cdot \Delta c = 0 \text{ | degree} = 3
$$

$$
(1 - n_0) \cdot (n_1^2 - n_1) = 0 \text{ | degree} = 6
$$

$$
(1 - n_0) \cdot (1 - n_1) \cdot \Delta a = 0 \text{ | degree} = 5
$$

The above constraints guarantee that when context changes, $n_0 = 1$. When context remains the same but address changes, $(1 - n_0) \cdot n_1 = 1$. And when neither the context nor the address change, $(1 - n_0) \cdot (1 - n_1) = 1$.

To enforce the values of the selector columns, we first require that they both contain only binary values.

$$
s_0^2 - s_0 = 0 \text{ | degree} = 2
$$

$$
s_1^2 - s_1 = 0 \text{ | degree} = 2
$$

Then we require that s_1 is always set to 1 during read operations when the context and address did not change and to 0 in all other cases.

$$
(1 - n_0) \cdot (1 - n_1) \cdot s'_0 \cdot (1 - s'_1) = 0 \text{ | degree} = 6
$$

$$
(n_0 + (1 - n_0) \cdot n_1 + (1 - s'_0)) \cdot s'_1 = 0 \text{ | degree} = 5
$$

The first constraint enforces that `s_1` is 1 when the operation is a read and `ctx` and `addr` are both unchanged. The second constraint enforces that when either the context changed, the address changed, or the operation is a write, then `s_1` is set to 0.

To enforce the values of context ID, address, and clock cycle grow monotonically as described in the previous section, we define the following constraint.

$$
\left(n_0 \cdot \Delta c + (1 - n_0) \cdot (n_1 \cdot \Delta a + (1 - n_1) \cdot \Delta i) \right) - (2^{16} \cdot d_1' + d_0') = 0 \text{ | degree} = 5
$$

Where $\Delta i = i' - i - 1$.

In addition to this constraint, we also need to make sure that values in registers d_0 and d_1 are less than 2^{16}, and this can be done with permutation-based range checks.

Next, we need to make sure that values at a given memory address are always initialized to 0. This can be done with the following constraint:

$$
s_0 \cdot (1 - s_1) \cdot v_i = 0 \text{ for } i \in \{0, 1, 2, 3\} \text{ | degree} = 3
$$

Thus, when the operation is a read and either the context changes or the address changes, values in the v_i columns are guaranteed to be zeros.

Lastly, we need to make sure that for the same context/address combination, the v_i columns of the current row are equal to the corresponding v_i columns of the next row. This can be done with the following constraints:

$$
s_1 \cdot (v_i' - v_i) = 0 \text{ for } i \in \{0, 1, 2, 3\} \text{ | degree} = 2
$$

Memory row value
Communication between the memory chiplet and the stack is accomplished via the chiplet bus b_{chip}. To respond to memory access requests from the stack, we need to divide the current value in b_{chip} by the value representing a row in the memory table. This value can be computed as follows:

$$
v_{mem} = \alpha_0 + \alpha_1 \cdot op_{mem} + \alpha_2 \cdot c + \alpha_3 \cdot a + \alpha_4 \cdot i + \sum_{j=0}^3(\alpha_{j + 5} \cdot v_j)
$$

Where, op_{mem} is the unique [operation label](./main.md#operation-labels) of the memory access operation.

To ensure that values of memory table rows are included into the chiplet bus, we impose the following constraint:

$$
b_{chip}' = b_{chip} \cdot v_{mem} \text{ | degree} = 2
$$

On the stack side, for every memory access request, a corresponding value is divided out of the b_{chip} column. Specifics of how this is done are described [here](../stack/io_ops.md#memory-access-operations).

---
id: constraints
title: AIR Constraints for Miden VM
sidebar_label: AIR Constraints
description: AIR constraint for Miden VM program decoder.
keywords:
  - docs
  - matic
  - polygon
  - wiki
  - miden
  - design
  - AIR
  - constraint
image: https://wiki.polygon.technology/img/thumbnail/polygon-miden.png
---

In this section we describe AIR constraint for Miden VM program decoder. These constraints enforce that the execution trace generated by the prover when executing a particular program complies with the rules described in the [previous section](./main.md).

To refer to decoder execution trace columns, we use the names shown on the diagram below (these are the same names as in the previous section). Additionally, we denote the register containing the value at the top of the stack as s_0.

![air_decoder_columns](../../assets/design/decoder/constraints/air_decoder_columns.png)

We assume that the VM exposes a flag per operation which is set to 1 when the operation is executed, and to 0 otherwise. The notation for such flags is f_{opname}. For example, when the VM executes a `PUSH` operation, flag $f_{push} = 1$. All flags are mutually exclusive - i.e., when one flag is set to 1 all other flags are set to 0. The flags are computed based on values in `op_bits` columns.

AIR constraints for the decoder involve operations listed in the table below. For each operation we also provide the degree of the corresponding flag and the effect that the operation has on the operand stack (however, in this section we do not cover the constraints needed to enforce the correct transition of the operand stack).

Operation	Flag	Degree	Effect on stack
`JOIN`	f_{join}	6	Stack remains unchanged.
`SPLIT`	f_{split}	6	Top stack element is dropped.
`LOOP`	f_{loop}	6	Top stack element is dropped.
`REPEAT`	f_{repeat}	4	Top stack element is dropped.
`SPAN`	f_{span}	6	Stack remains unchanged.
`RESPAN`	f_{respan}	4	Stack remains unchanged.
`CALL`	f_{call}	4	Top stack element is dropped.
`SYSCALL`	$f_{syscall}$	4	Top stack element is dropped.
`END`	f_{end}	4	When exiting a loop block, top stack element is dropped; otherwise, the stack remains unchanged.
`HALT`	f_{halt}	4	Stack remains unchanged.
`PUSH`	f_{push}	4	An immediate value is pushed onto the stack.

We also use the [control flow flag](../stack/op_constraints.md#control-flow-flag) f_{ctrl} exposed by the VM, which is set when any one of the above control flow operations is being executed. It has degree 4.

As described [previously](./main.md#program-decoding), the general idea of the decoder is that the prove provides the program to the VM by populating some of cells in the trace non-deterministically. Values in these are then used to update virtual tables (represented via multiset checks) such as block hash table, block stack table etc. Transition constraints are used to enforce that the tables are updates correctly, and we also apply boundary constraints to enforce the correct initial and final states of these tables. One of these boundary constraints binds the execution trace to the hash of the program being executed. Thus, if the virtual tables were updated correctly and boundary constraints hold, we can be convinced that the prover executed the claimed program on the VM.

In the sections below, we describe constraints according to their logical grouping. However, we start out with a set of general constraints which are applicable to multiple parts of the decoder.

General constraints

When `SPLIT` or `LOOP` operation is executed, the top of the operand stack must contain a binary value:

$$

$$
(f_{split} + f_{loop}) \cdot (s_0^2 - s_0) = 0 \text{ | degree} = 8
$$

When `REPEAT` operation is executed, the value at the top of the operand stack must be 1:

$$
f_{repeat} \cdot (1 - s_0) = 0 \text{ | degree} = 5
$$

Also, when `REPEAT` operation is executed, the value in h_4 column (the `is_loop_body` flag), must be set to 1. This ensures that `REPEAT` operation can be executed only inside a loop:

$$
f_{repeat} \cdot (1 - h_4) = 0 \text{ | degree} = 5
$$

When `RESPAN` operation is executed, we need to make sure that the block ID is incremented by 8:

$$
f_{respan} \cdot (a' - a - 8) = 0 \text{ | degree} = 5
$$

When `END` operation is executed and we are exiting a *loop* block (i.e., `is_loop`, value which is stored in h_5, is 1), the value at the top of the operand stack must be 0:

$$
f_{end} \cdot h_5 \cdot s_0 = 0 \text{ | degree} = 6
$$

Also, when `END` operation is executed and the next operation is `REPEAT`, values in $h_0, ..., h_4$ (the hash of the current block and the `is_loop_body` flag) must be copied to the next row:

$$
f_{end} \cdot f_{repeat}' \cdot (h_i' - h_i) = 0 \text { for } i \in 0..4 \text{ | degree} = 9
$$

A `HALT` instruction can be followed only by another `HALT` instruction:

$$
f_{halt} \cdot (1 - f_{halt}') = 0 \text{ | degree} = 8
$$

When a `HALT` operation is executed, block address column must be 0:

$$
f_{halt} \cdot a = 0 \text{ | degree} = 5
$$

Values in `op_bits` columns must be binary (i.e., either 1 or 0):

$$
b_i^2 - b_i = 0 \text{ for } i \in 0..6 \text{ | degree} = 2
$$

When the value in `is_span` column is set to 1, control flow operations cannot be executed on the VM, but when `is_span` flag is 0, only control flow operations can be executed on the VM:

$$
1 - sp - f_{ctrl} = 0 \text{ | degree} = 4
$$

Block hash computation constraints
As described [previously](./main.md#program-block-hashing), when the VM starts executing a new block, it also initiates computation of the block's hash. There are two separate methodologies for computing block hashes.

For *join*, *split*, and *loop* blocks, the hash is computed directly from the hashes of the block's children. The prover provides these child hashes non-deterministically by populating registers $h_0,..., h_7$. The hasher is initialized using the hash chiplet, and we use the address of the hasher as the block's ID. The result of the hash is available 7 rows down in the hasher table (i.e., at row with index equal to block ID plus 7). We read the result from the hasher table at the time the `END` operation is executed for a given block.

For *span* blocks, the hash is computed by absorbing a linear sequence of instructions (organized into operation groups and batches) into the hasher and then returning the result. The prover provides operation batches non-deterministically by populating registers $h_0, ..., h_7$. Similarly to other blocks, the hasher is initialized using the hash chiplet at the start of the block, and we use the address of the hasher as the ID of the first operation batch in the block. As we absorb additional operation batches into the hasher (by executing `RESPAN` operation), the batch address is incremented by 8. This moves the "pointer" into the hasher table 8 rows down with every new batch. We read the result from the hasher table at the time the `END` operation is executed for a given block.

The way the decoder communicates with the hash chiplet is by dividing values of the multiset check column p_0 by the values of operations providing inputs to or reading outputs from the hash chiplet. A constraint to enforce this would look as $p_0' \cdot u = p_0$, where u is the value which defines the operation.

In constructing value of u for decoder AIR constraints, we will use the following labels (see [here](../chiplets/hasher.md#multiset-check-constraints) for an explanation of how values for these labels are computed):

* m_{bp} this label specifies that we are starting a new hash computation.
* m_{abp} this label specifies that we are absorbing the next sequence of 8 elements into an ongoing hash computation.
* m_{hout} this label specifies that we are reading the result of a hash computation.

To simplify constraint description, we define the following variables:

$$
h_{init} = \alpha_0 + \alpha_1 \cdot m_{bp} + \alpha_2 \cdot a' + \sum_{i=0}^7(\alpha_{i + 8} \cdot h_i)
$$

In the above, h_{init} can be thought of as initiating a hasher with address a' and absorbing 8 elements from the hasher state ($h_0, ..., h_7$) into it. Control blocks are always padded to fill the hasher rate and as such the α_4 (first capacity register) term is set to 0.

$$
h_{abp} = \alpha_0 + \alpha_1 \cdot m_{abp} + \alpha_2 \cdot a' + \sum_{i=0}^7(\alpha_{i + 8} \cdot h_i)
$$

It should be noted that a refers to a column in the decoder, as depicted. The addresses in this column are set using the address from the hasher chiplet for the corresponding hash initialization / absorption / return. In the case of h_{abp} the value of the address in column a in the current row of the decoder is set to equal the value of the address of the row in the hasher chiplet where the previous absorption (or initialization) occurred. a' is the address of the next row of the decoder, which is set to equal the address in the hasher chiplet where the absorption referred to by the h_{abp} label is happening.

$$
h_{res} = \alpha_0 + \alpha_1 \cdot m_{hout} + \alpha_2 \cdot (a + 7) + \sum_{i=0}^3(\alpha_{i + 8} \cdot h_i)
$$

In the above, a represents the address value in the decoder which corresponds to the hasher chiplet address at which the hasher was initialized (or the last absorption took place). As such, $a + 7$ corresponds to the hasher chiplet address at which the result is returned.

$$
f_{ctrli} = f_{join} + f_{split} + f_{loop} + f_{call} + f_{syscall} \text{ | degree} = 6
$$

In the above, f_{ctrli} is set to 1 when a control flow operation that signifies the initialization of a control block is being executed on the VM. Otherwise, it is set to 0.

$$
d = \sum_{b=0}^6(b_i \cdot 2^i)
$$

In the above, d represents the opcode value of the opcode being executed on the virtual machine. It is calculated via a bitwise combination of the op bits. We leverage the opcode value to achieve domain separation when hashing control blocks. This is done by populating the second capacity register of the hasher with the value d via the α_5 term when initializing the hasher.

Using the above variables, we define operation values as described below.

When a control block initializer operation (`JOIN`, `SPLIT`, `LOOP`, `CALL`, `SYSCALL`) is executed, a new hasher is initialized and the contents of $h_0, ..., h_7$ are absorbed into the hasher. As mentioned above, the opcode value d is populated in the second capacity resister via the α_5 term.

$$
u_{ctrli} = f_{ctrli} \cdot (h_{init} + \alpha_5 \cdot d) \text{ | degree} = 7
$$

When `SPAN` operation is executed, a new hasher is initialized and contents of $h_0, ..., h_7$ are absorbed into the hasher. The number of operation groups to be hashed is padded to a multiple of the rate width (8) and so the α_4 is set to 0:

$$
u_{span} = f_{span} \cdot h_{init} \text{ | degree} = 7
$$

When `RESPAN` operation is executed, contents of $h_0, ..., h_7$ (which contain the new operation batch) are absorbed into the hasher:

$$
u_{respan} = f_{respan} \cdot h_{abp} \text{ | degree} = 5
$$

When `END` operation is executed, the hash result is copied into registers $h_0, .., h_3$:

$$
u_{end} = f_{end} \cdot h_{res} \text{ | degree} = 5
$$

Using the above definitions, we can describe the constraint for computing block hashes as follows:

$$
p_0' \cdot (u_{ctrli} + u_{span} + u_{respan} + u_{end} + \\
1 - (f_{ctrli} + f_{span} + f_{respan} + f_{end})) = p_0
$$

We need to add 1 and subtract the sum of the relevant operation flags to ensure that when none of the flags is set to 1, the above constraint reduces to $p_0' = p_0$.

The degree of this constraint is 8.

Block stack table constraints
As described [previously](./main.md#block-stack-table), block stack table keeps track of program blocks currently executing on the VM. Thus, whenever the VM starts executing a new block, an entry for this block is added to the block stack table. And when execution of a block completes, it is removed from the block stack table.

Adding and removing entries to/from the block stack table is accomplished as follows:
* To add an entry, we multiply the value in column p_1 by a value representing a tuple `(blk_id, prnt_id, is_loop)`. A constraint to enforce this would look as $p_1' = p_1 \cdot v$, where v is the value representing the row to be added.
* To remove an entry, we divide the value in column p_1 by a value representing a tuple `(blk_id, prnt_id, is_loop)`. A constraint to enforce this would look as $p_1' \cdot u = p_1$, where u is the value representing the row to be removed.

Before describing the constraints for the block stack table, we first describe how we compute the values to be added and removed from the table for each operation. In the below, for block start operations (`JOIN`, `SPLIT`, `LOOP`, `SPAN`) a refers to the ID of the parent block, and a' refers to the ID of the starting block. For `END` operation, the situation is reversed: a is the ID of the ending block, and a' is the ID of the parent block. For `RESPAN` operation, a refers to the ID of the current operation batch, a' refers to the ID of the next batch, and the parent ID for both batches is set by the prover non-deterministically in register h_1.

When `JOIN` operation is executed, row $(a', a, 0)$ is added to the block stack table:

$$
v_{join} = f_{join} \cdot (\alpha_0 + \alpha_1 \cdot a' + \alpha_2 \cdot a) \text{ | degree} = 7
$$

When `SPLIT` operation is executed, row $(a', a, 0)$ added to the block stack table:

$$
v_{split} = f_{split} \cdot (\alpha_0 + \alpha_1 \cdot a' + \alpha_2 \cdot a) \text{ | degree} = 7
$$

When `LOOP` operation is executed, row $(a', a, 1)$ is added to the block stack table if the value at the top of the operand stack is 1, and row $(a', a, 0)$ is added to the block stack table if the value at the top of the operand stack is 0:

$$
v_{loop} = f_{loop} \cdot (\alpha_0 + \alpha_1 \cdot a' + \alpha_2 \cdot a + \alpha_3 \cdot s_0) \text{ | degree} = 7
$$

When `SPAN` operation is executed, row $(a', a, 0)$ is added to the block stack table:

$$
v_{span} = f_{span} \cdot (\alpha_0 + \alpha_1 \cdot a' + \alpha_2 \cdot a) \text{ | degree} = 7
$$

When `RESPAN` operation is executed, row $(a, h_1', 0)$ is removed from the block stack table, and row $(a', h_1', 0)$ is added to the table. The prover sets the value of register h_1 at the next row to the ID of the parent block:

$$
u_{respan} = f_{respan} \cdot (\alpha_0 + \alpha_1 \cdot a + \alpha_2 \cdot h_1') \text{ | degree} = 5 \\
v_{respan} = f_{respan} \cdot (\alpha_0 + \alpha_1 \cdot a' + \alpha_2 \cdot h_1') \text{ | degree} = 5
$$

When `END` operation is executed, row (a, a', h_5) is removed from the block span table. Register h_5 contains the `is_loop` flag:

$$
u_{end} = f_{end} \cdot (\alpha_0 + \alpha_1 \cdot a + \alpha_2 \cdot a' + \alpha_3 \cdot h_5) \text{ | degree} = 5
$$

Using the above definitions, we can describe the constraint for updating the block stack table as follows:

$$
p_1' \cdot (u_{end} + u_{respan} + 1 - (f_{end} + f_{respan})) = \\
p_1 \cdot (v_{join} + v_{split} + v_{loop} + v_{span} + v_{respan} + 1 - (f_{join} + f_{split} + f_{loop} + f_{span} + f_{respan}
))
$$

We need to add 1 and subtract the sum of the relevant operation flags from each side to ensure that when none of the flags is set to 1, the above constraint reduces to $p_1' = p_1$.

The degree of this constraint is 8.

In addition to the above transition constraint, we also need to impose boundary constraints against the p_1 column to make sure the first and the last value in the column is set to 1. This enforces that the block stack table starts and ends in an empty state.

Block hash table constraints
As described [previously](./main.md#block-hash-table), when the VM starts executing a new program block, it adds hashes of the block's children to the block hash table. And when the VM finishes executing a block, it removes the block's hash from the block hash table. This means that the block hash tables gets updated when we execute `JOIN`, `SPLIT`, `LOOP`, `REPEAT`, and `END` operations (executing `SPAN` operation does not affect the block hash table because a *span* block has no children).

Adding and removing entries to/from the block hash table is accomplished as follows:
* To add an entry, we multiply the value in column p_2 by a value representing a tuple `(prnt_id, block_hash, is_first_child, is_loop_body)`. A constraint to enforce this would look as $p_2' = p_2 \cdot v$, where v is the value representing the row to be added.
* To remove an entry, we divide the value in column p_2 by a value representing a tuple `(prnt_id, block_hash, is_first_child, is_loop_body)`. A constraint to enforce this would look as $p_2' \cdot u = p_2$, where u is the value representing the row to be removed.

To simplify constraint descriptions, we define values representing left and right children of a block as follows:

$$
ch_1 = \alpha_0 + \alpha_1 \cdot a' + \sum_{i=0}^3(a_{i+2} \cdot h_i) \text{ | degree} = 1 \\
ch_2 = \alpha_0 + \alpha_1 \cdot a' + \sum_{i=0}^3(a_{i+2} \cdot h_{i+4}) \text{ | degree} = 1
$$

Graphically, this looks like so:

![air_decoder_left_right_child](../../assets/design/decoder/constraints/air_decoder_left_right_child.png)

In a similar manner, we define a value representing the result of hash computation as follows:

$$
bh = \alpha_0 + \alpha_1 \cdot a + \sum_{i=0}^3(\alpha_{i+2} \cdot h_i) + \alpha_7 \cdot h_4 \text{ | degree} = 1
$$

Note that in the above we use a (block address from the current row) rather than a' (block address from the next row) as we did for for values of ch_1 and ch_2. Also, note that we are not adding a flag indicating whether the block is the first child of a join block (i.e., α_6 term is missing). It will be added later on.

Using the above variables, we define row values to be added to and removed from the block hash table as follows.

When `JOIN` operation is executed, hashes of both child nodes are added to the block hash table. We add α_6 term to the first child value to differentiate it from the second child (i.e., this sets `is_first_child` to 1):

$$
v_{join} = f_{join} \cdot (ch_1 + \alpha^6) \cdot ch_2 \text{ | degree} = 8
$$

When `SPLIT` operation is executed and the top of the stack is 1, hash of the *true* branch is added to the block hash table, but when the top of the stack is 0, hash of the *false* branch is added to the block hash table:

$$
v_{split} = f_{split} \cdot (s_0 \cdot ch_1 + (1 - s_0) \cdot ch_2) \text{ | degree} = 8
$$

When `LOOP` operation is executed and the top of the stack is 1, hash of loop body is added to the block hash table. We add α_7 term to indicate that the child is a body of a loop. The below also means that if the top of the stack is 0, nothing is added to the block hash table as the expression evaluates to 0:

$$
v_{loop} = f_{loop} \cdot s_0 \cdot (ch_1 + \alpha_7) \text{ | degree} = 8
$$

When `REPEAT` operation is executed, hash of loop body is added to the block hash table. We add α_7 term to indicate that the child is a body of a loop:

$$v_{repeat} = f_{repeat} \cdot (ch_1 + \alpha_7) \text{ | } \text{degree} = 5$$

When `END` operation is executed, hash of the completed block is removed from the block hash table. However, we also need to differentiate between removing the first and the second child of a *join* block. We do this by looking at the next operation. Specifically, if the next operation is neither `END` nor `REPEAT` we know that another block is about to be executed, and thus, we have just finished executing the first child of a *join* block. Thus, if the next operation is neither `END` nor `REPEAT` we need to set the term for α_6 coefficient to 1 as shown below:

$$
u_{end} = f_{end} \cdot (bh + \alpha_6 \cdot (1 - (f_{end}' + f_{repeat}'))) \text{ | } \text{degree} = 8
$$

Using the above definitions, we can describe the constraint for updating the block hash table as follows:

$$
p_2' \cdot (u_{end} + 1 - f_{end}) = \\
p_2 \cdot (v_{join} + v_{split} + v_{loop} + v_{repeat} + 1 - (f_{join} + f_{split} + f_{loop} + f_{repeat}))
$$

We need to add 1 and subtract the sum of the relevant operation flags from each side to ensure that when none of the flags is set to 1, the above constraint reduces to $p_2' = p_2$.

The degree of this constraint is 9.

In addition to the above transition constraint, we also need to set the following boundary constraints against the p_2 column:
* The first value in the column represents a row for the entire program. Specifically, the row tuple would be `(0, program_hash, 0, 0)`. This row should be removed from the table when the last `END` operation is executed.
* The last value in the column is 1 - i.e., the block hash table is empty.

Span block
Span block constraints ensure proper decoding of span blocks. In addition to the block stack table constraints and block hash table constraints described previously, decoding of span blocks requires constraints described below.

In-span column constraints
The `in_span` column (denoted as sp) is used to identify rows which execute non-control flow operations. The values in this column are set as follows:

* Executing a `SPAN` operation sets the value of `in_span` column to 1.
* The value remains 1 until the `END` operation is executed.
* If `RESPAN` operation is executed between `SPAN` and `END` operations, in the row at which `RESPAN` operation is executed `in_span` is set to 0. It is then reset to 1 in the following row.
* In all other cases, value in the `in_span` column should be 0.

The picture below illustrates the above rules.

![air_decoder_in_spans_column_constraint](../../assets/design/decoder/constraints/air_decoder_in_spans_column_constraint.png)

To enforce the above rules we need the following constraints.

When executing `SPAN` or `RESPAN` operation, the next value in sp column must be set to 1:

$$
(f_{span} + f_{respan}) \cdot (1 - sp') = 0 \text{ | degree} = 7
$$

When the next operation is `END` or `RESPAN`, the next value in sp column must be set 0.

$$
(f_{end}' + f_{respan}') \cdot sp' = 0 \text{ | degree} = 5
$$

In all other cases, the value in sp column must be copied over to the next row:

$$
(1 - f_{span} - f_{respan} - f_{end}' - f_{respan}') \cdot (sp' - sp) = 0 \text{ | degree} = 7
$$

Additionally, we will need to impose a boundary constraint which specifies that the first value in $sp = 0$. Note, however, that we do not need to impose a constraint ensuring that values in sp are binary - this will follow naturally from the above constraints.

Also, note that the combination of the above constraints makes it impossible to execute `END` or `RESPAN` operations right after `SPAN` or `RESPAN` operations.

Block address constraints
When we are inside a *span* block, values in block address columns (denoted as a) must remain the same. This can be enforced with the following constraint:

$$
sp \cdot (a' - a) = 0 \text{ | degree} = 2
$$

Notice that this constraint does not apply when we execute any of the control flow operations. For such operations, the prover sets the value of the a column non-deterministically, except for the `RESPAN` operation. For the `RESPAN` operation the value in the a column is incremented by 8, which is enforced by a constraint described previously.

Notice also that this constraint implies that when the next operation is the `END` operation, the value in the a column must also be copied over to the next row. This is exactly the behavior we want to enforce so that when the `END` operation is executed, the block address is set to the address of the current span batch.

Group count constraints
The `group_count` column (denoted as gc) is used to keep track of the number of operation groups which remains to be executed in a span block.

In the beginning of a span block (i.e., when `SPAN` operation is executed), the prover sets the value of gc non-deterministically. This value is subsequently decremented according to the rules described below. By the time we exit the span block (i.e., when `END` operation is executed), the value in gc must be 0.

The rules for decrementing values in the gc column are as follows:
* The count cannot be decremented by more than 1 in a single row.
* When an operation group is fully executed (which happens when $h_0 = 0$ inside a span block), the count is decremented by 1.
* When `SPAN`, `RESPAN`, or `PUSH` operations are executed, the count is decremented by 1.

Note that these rules imply that `PUSH` operation cannot be the last operation in an operation group (otherwise the count would have to be decremented by 2).

To simplify the description of the constraints, we will define the following variable:

$$
\Delta gc = gc - gc'
$$

Using this variable, we can describe the constraints against the gc column as follows:

Inside a *span* block, group count can either stay the same or decrease by one:

$$
sp \cdot \Delta gc \cdot (\Delta gc - 1) = 0 \text{ | degree} = 3
$$

When group count is decremented inside a *span* block, either h_0 must be 0 (we consumed all operations in a group) or we must be executing `PUSH` operation:

$$
sp \cdot \Delta gc \cdot (1 - f_{push})\cdot h_0 = 0 \text{ | degree} = 7
$$

Notice that the above constraint does not preclude $f_{push} = 1$ and $h_0 = 0$ from being true at the same time. If this happens, op group decoding constraints (described [here](#op-group-decoding-constraints)) will force that the operation following the `PUSH` operation is a `NOOP`.

When executing a `SPAN`, a `RESPAN`, or a `PUSH` operation, group count must be decremented by 1:

$$
(f_{span} + f_{respan} + f_{push}) \cdot (\Delta gc - 1) = 0 \text{ | degree} = 7
$$

If the next operation is either an `END` or a `RESPAN`, group count must remain the same:

$$
\Delta gc \cdot (f_{end}' + f_{respan}') = 0 \text{ | degree} = 5
$$

When an `END` operation is executed, group count must be 0:

$$
f_{end} \cdot gc = 0 \text{ | degree} = 5
$$

Op group decoding constraints
Inside a *span* block, register h_0 is used to keep track of operations to be executed in the current operation group. The value of this register is set by the prover non-deterministically at the time when the prover executes a `SPAN` or a `RESPAN` operation, or when processing of a new operation group within a batch starts. The picture below illustrates this.

![air_decoder_op_group_constraint](../../assets/design/decoder/constraints/air_decoder_op_group_constraint.png)

In the above:
* The prover sets the value of h_0 non-deterministically at row 0. The value is set to an operation group containing operations `op0` through `op8`.
* As we start executing the group, at every row we "remove" the least significant operation from the group. This can be done by subtracting opcode of the operation from the group, and then dividing the result by 2^7.
* By row 9 the group is fully executed. This decrements the group count and set `op_index` to 0 (constraints against `op_index` column are described in the next section).
* At row 10 we start executing the next group with operations `op9` through `op11`. In this case, the prover populates h_0 with the group having its first operation (`op9`) already removed, and sets the `op_bits` registers to the value encoding `op9`.
* By row 12 this group is also fully executed.

To simplify the description of the constraints, we define the following variables:

$$
op = \sum_{i=0}^6 (b_i \cdot 2^i) \\
f_{sgc} = sp \cdot sp' \cdot (1 - \Delta gc)
$$

op is just an opcode value implied by the values in `op_bits` registers. f_{sgc} is a flag which is set to 1 when the group count within a *span* block does not change. We multiply it by sp' to make sure the flag is 0 when we are about to end decoding of an operation batch. Note that f_{sgc} flag is mutually exclusive with f_{span}, f_{respan}, and f_{push} flags as these three operations decrement the group count.

Using these variables, we can describe operation group decoding constraints as follows:

When a `SPAN`, a `RESPAN`, or a `PUSH` operation is executed or when the group count does not change, the value in h_0 should be decremented by the value of the opcode in the next row.

$$
(f_{span} + f_{respan} + f_{push} + f_{sgc}) \cdot (h_0 - h_0' \cdot 2^7 - op') = 0 \text{ | degree} = 7
$$

Notice that when the group count does change, and we are not executing f_{span}, f_{respan}, or f_{push} operations, no constraints are placed against h_0, and thus, the prover can populate this register non-deterministically.

When we are in a *span* block and the next operation is `END` or `RESPAN`, the current value in h_0 column must be 0.

$$
sp \cdot (f_{end}' + f_{respan}') \cdot h_0 = 0 \text{ | degree} = 6
$$

Op index constraints
The `op_index` column (denoted as ox) tracks index of an operation within its operation group. It is used to ensure that the number of operations executed per group never exceeds 9. The index is zero-based, and thus, the possible set of values for ox is between 0 and 8 (both inclusive).

To simplify the description of the constraints, we will define the following variables:

$$
ng = \Delta gc - f_{push} \\
\Delta ox = ox' - ox
$$

The value of ng is set to 1 when we are about to start executing a new operation group (i.e., group count is decremented but we did not execute a `PUSH` operation). Using these variables, we can describe the constraints against the ox column as follows.

When executing `SPAN` or `RESPAN` operations the next value of `op_index` must be set to 0:

$$
(f_{span} + f_{respan}) \cdot ox' = 0 \text{ | degree} = 7
$$

When starting a new operation group inside a *span* block, the next value of `op_index` must be set to 0. Note that we multiply by sp to exclude the cases when the group count is decremented because of `SPAN` or `RESPAN` operations:

$$
sp \cdot ng \cdot ox' = 0 \text{ | degree} = 6
$$

When inside a *span* block but not starting a new operation group, `op_index` must be incremented by 1. Note that we multiply by sp' to exclude the cases when we are about to exit processing of an operation batch (i.e., the next operation is either `END` or `RESPAN`):

$$
sp \cdot sp' \cdot (1 - ng) \cdot (\Delta ox - 1) = 0 \text{ | degree} = 7
$$

Values of `op_index` must be in the range $[0, 8]$.

$$
\prod_{i=0}^{8}(ox - i) = 0 \text{ | degree} = 9
$$

Op batch flags constraints
Operation batch flag columns (denoted bc_0, bc_1, and bc_2) are used to specify how many operation groups are present in an operation batch. This is relevant for the last batch in a span block (or the first batch if there is only one batch in a block) as all other batches should be completely full (i.e., contain 8 operation groups).

These columns are used to define the following 4 flags:

* $f_{g8} = bc_0$: there are 8 operation groups in the batch.
* $f_{g4} = (1 - bc_0) \cdot bc_1 \cdot bc_2$: there are 4 operation groups in the batch.
* $f_{g2} = (1 - bc_0) \cdot (1 - bc_1) \cdot bc_2$: there are 2 operation groups in the batch.
* $f_{g1} = (1 - bc_0) \cdot bc_1 \cdot (1 - bc_2)$: there is only 1 operation groups in the batch.

Notice that the degree of f_{g8} is 1, while the degree of the remaining flags is 3.

These flags can be set to 1 only when we are executing `SPAN` or `RESPAN` operations as this is when the VM starts processing new operation batches. Also, for a given flag we need to ensure that only the specified number of operations groups are present in a batch. This can be done with the following constraints.

All batch flags must be binary:

$$
bc_i^2 - bc_i = 0 \text{ for } i \in 0..2 \text{ | degree} = 2
$$

When `SPAN` or `RESPAN` operations is executed, one of the batch flags must be set to 1.

$$
(f_{span} + f_{respan}) - (f_{g1} + f_{g2} + f_{g4} + f_{g8}) = 0 \text{ | degree} = 6
$$

When we have at most 4 groups in a batch, registers $h_4, ..., h_7$ should be set to 0's.

$$
(f_{g1} + f_{g2} + f_{g4}) \cdot h_i = 0 \text{ for } i \in 4..7 \text{ | degree} = 4
$$

When we have at most 2 groups in a batch, registers h_2 and h_3 should also be set to 0's.

$$
(f_{g1} + f_{g2}) \cdot h_i = 0 \text{ for } i \in 2, 3 \text{ | degree} = 4
$$

When we have at most 1 groups in a batch, register h_1 should also be set to 0.

$$
f_{g1} \cdot h_1 = 0 \text{ | degree} = 4
$$

Op group table constraints
Op group table is used to ensure that all operation groups in a given batch are consumed before a new batch is started (i.e., via a `RESPAN` operation) or the execution of a *span* block is complete (i.e., via an `END` operation). The op group table is updated according to the following rules:

* When a new operation batch is started, we add groups from this batch to the table. To add a group to the table, we multiply the value in column p_3 by a value representing a tuple `(batch_id, group_pos, group)`. A constraint to enforce this would look as $p_3' = p_3 \cdot v$, where v is the value representing the row to be added. Depending on the batch, we may need to add multiple groups to the table (i.e., $p_3' = p_3 \cdot v_1 \cdot v_2 \cdot v_3 ...$). Flags f_{g1}, f_{g2}, f_{g4}, and f_{g8} are used to define how many groups to add.
* When a new operation group starts executing or when an immediate value is consumed, we remove the corresponding group from the table. To do this, we divide the value in column p_3 by a value representing a tuple `(batch_id, group_pos, group)`. A constraint to enforce this would look as $p_3' \cdot u = p_3$, where u is the value representing the row to be removed.

To simplify constraint descriptions, we first define variables representing the rows to be added to and removed from the op group table.

When a `SPAN` or a `RESPAN` operation is executed, we compute the values of the rows to be added to the op group table as follows:

$$
v_i = \alpha_0 + \alpha_1 \cdot a' + \alpha_2 \cdot (gc - i) + \alpha_3 \cdot h_{i} \text{ | degree} = 1
$$

Where $i \in 1..7$. Thus, v_1 defines row value for group in h_1, v_2 defines row value for group h_2 etc. Note that batch address column comes from the next row of the block address column (a').

We compute the value of the row to be removed from the op group table as follows:

$$
u = \alpha_0 + \alpha_1 \cdot a + \alpha^2 \cdot gc + \alpha_3 \cdot ((h_0' \cdot 2^7 + op') \cdot (1 - f_{push}) + s_0' \cdot f_{push}) \text{ | degree} = 5
$$

In the above, the value of the group is computed as $(h_0' \cdot 2^7 + op') \cdot (1 - f_{push}) + s_0' \cdot f_{push}$. This basically says that when we execute a `PUSH` operation we need to remove the immediate value from the table. This value is at the top of the stack (column s_0) in the next row. However, when we are not executing a `PUSH` operation, the value to be removed is an op group value which is a combination of values in h_0 and `op_bits` columns (also in the next row). Note also that value for batch address comes from the current value in the block address column (a), and the group position comes from the current value of the group count column (gc).

We also define a flag which is set to 1 when a group needs to be removed from the op group table.

$$
f_{dg} = sp \cdot \Delta gc
$$

The above says that we remove groups from the op group table whenever group count is decremented. We multiply by sp to exclude the cases when the group count is decremented due to `SPAN` or `RESPAN` operations.

Using the above variables together with flags f_{g2}, f_{g4}, f_{g8} defined in the previous section, we describe the constraint for updating op group table as follows (note that we do not use f_{g1} flag as when a batch consists of a single group, nothing is added to the op group table):

$$
p_3' \cdot (f_{dg} \cdot u + 1 - f_{dg}) = p_3 \cdot (f_{g2} \cdot v_1 + f_{g4} \cdot \prod_{i=1}^3 v_i + f_{g8} \cdot \prod_{i=1}^7 v_i - 1 + (f_{span} + f_{respan}))
$$

The above constraint specifies that:
* When `SPAN` or `RESPAN` operations are executed, we add between 1 and 7 groups to the op group table.
* When group count is decremented inside a *span* block, we remove a group from the op group table.

The degree of this constraint is 9.

In addition to the above transition constraint, we also need to impose boundary constraints against the p_3 column to make sure the first and the last value in the column is set to 1. This enforces that the op group table table starts and ends in an empty state.
---
id: main
title: Miden VM Program Decoder
sidebar_label: Overview
description: "How Miden VM decoder works."
keywords:
  - docs
  - matic
  - polygon
  - wiki
  - miden
  - design
  - decoder
image: https://wiki.polygon.technology/img/thumbnail/polygon-miden.png
---

Miden VM program decoder is responsible for ensuring that a program with a given [MAST root](../programs.md) is executed by the VM. As the VM executes a program, the decoder does the following:

1. Decodes a sequence of field elements supplied by the prover into individual operation codes (or *opcodes* for short).
2. Organizes the sequence of field elements into code blocks, and computes the hash of the program according to the methodology described [here](../programs.md#program-hash-computation).

At the end of program execution, the decoder outputs the computed program hash. This hash binds the sequence of opcodes executed by the VM to a program the prover claims to have executed. The verifier uses this hash during the STARK proof verification process to verify that the proof attests to a correct execution of a specific program (i.e., the prover didn't claim to execute program A while in fact executing a different program B).

The sections below describe how Miden VM decoder works. Throughout these sections we make the following assumptions:

1. An opcode requires 7 bits to represent.
2. An immediate value requires one full field element to represent.
3. A `NOOP` operation has a numeric value of 0, and thus, can be encoded as seven zeros. Executing a `NOOP` operation does not change the state of the VM, but it does advance operation counter, and may affect program hash.

Program execution

```
Miden VM programs consist of a set of code blocks organized into a binary tree. The leaves of the tree contain linear sequences of
instructions, and control flow is defined by the internal nodes of the tree.

Managing control flow in the VM is accomplished by executing control flow operations listed in the table below. Each of these
operations require exactly one VM cycle to execute.

| Operation | Description |
| --------- | ----------- |
| `JOIN`    | Initiates processing of a new [Join block](../programs.md#join-block). |
| `SPLIT`   | Initiates processing of a new [Split block](../programs.md#split-block). |
| `LOOP`    | Initiates processing of a new [Loop block](../programs.md#loop-block). |
| `REPEAT`  | Initiates a new iteration of an executing loop. |
| `SPAN`    | Initiates processing of a new [Span block](../programs.md#span-block). |
| `RESPAN`  | Initiates processing of a new operation batch within a span block. |
| `CALL`    | Initiates processing of a new [Call block](../programs.md#call-block). |
| `SYSCALL` | Initiates processing ofa new  [Syscall block](../programs.md#syscall-block). |
| `END`     | Marks the end of a program block. |
| `HALT`    | Marks the end of the entire program. |

Let's consider a simple program below:
```

begin if.true else end end

```
Block structure of this program is shown below.
```

JOIN SPAN END SPLIT SPAN END SPAN END END END

```
Executing this program on the VM can result in one of two possible instruction sequences. First, if after operations in
`<operations1>` are executed the top of the stack is $1$, the VM will execute the following:
```

JOIN SPAN END SPLIT SPAN END END END HALT

```
However, if after `<operations1>` are executed, the top of the stack is $0$, the VM will execute the following:
```

JOIN SPAN END SPLIT SPAN END END END HALT

```
The main task of the decoder is to output exactly the same program hash, regardless of which one of the two possible execution
paths was taken. However, before we can describe how this is achieved, we need to give an overview of the overall decoder
structure.

## Decoder structure

The decoder is one of the more complex parts of the VM. It consists of the following components:

* Main [execution trace](#decoder-trace) consisting of $24$ trace columns which contain the state of the decoder at a given cycle
of a computation.
* Connection to the hash chiplet, which is used to offload [hash computations](#program-block-hashing) from the decoder.
* $3$ [virtual tables](#control-flow-tables) (implemented via multi-set checks), which keep track of code blocks and operations
executing on the VM.

### Decoder trace

Decoder trace columns can be grouped into several logical sets of registers as illustrated below.

![decoder_trace.png](../../assets/design/decoder/decoder_trace.png)

These registers have the following meanings:

1. Block address register $a$. This register contains address of the hasher for the current block (row index from the auxiliary
hashing table). It also serves the role of unique block identifiers. This is convenient, because hasher addresses are guaranteed
to be unique.
2. Registers $b_0, ..., b_6$, which encode opcodes for operation to be executed by the VM. Each of these registers can contain a
single binary value (either $1$ or $0$). And together these values describe a single opcode.
3. Hasher registers $h_0, ..., h_7$. When control flow operations are executed, these registers are used to provide inputs for the
current block's hash computation (e.g., for `JOIN`, `SPLIT`, `LOOP`, `SPAN`, `CALL`, `SYSCALL` operations) or to record the result
of the hash computation (i.e., for `END` operation). However, when regular operations are executed, $2$ of these registers are
used to help with op group decoding, and the remaining $6$ can be used to hold operation-specific helper variables.
4. Register $sp$ which contains a binary flag indicating whether the VM is currently executing instructions inside a *span* block.
The flag is set to $1$ when the VM executes non-control flow instructions, and is set to $0$ otherwise.
5. Register $gc$ which keep track of the number of unprocessed operation groups in a given *span* block.
6. Register $ox$ which keeps track of a currently executing operation's index within its operation group.
7. Operation batch flags $c_0, c_1, c_2$ which indicate how many operation groups a given operation batch contains. These flags
are set only for `SPAN` and `RESPAN` operations, and are set to $0$'s otherwise.
```

8. Two additional registers (not shown) used primarily for constraint degree reduction.

Program block hashing

To compute hashes of program blocks, the decoder relies on the [hash chiplet](../chiplets/hasher.md). Specifically, the decoder needs to perform two types of hashing operations:

1. A simple 2-to-1 hash, where we provide a sequence of 8 field elements, and get back 4 field elements representing the result. Computing such a hash requires 8 rows in the hash chiplet.
2. A sequential hash of n elements. Computing such a hash requires multiple absorption steps, and at each step 8 field elements are absorbed into the hasher. Thus, computing a sequential hash of n elements requires $\lceil {n/8} \rceil$ rows in the hash chiplet. At the end, we also get 4 field elements representing the result.

We denote the running product column used to keep track of hash chiplet state as p_0. To make hashing requests to the hash chiplet and to read the results from it, we will need to divide out relevant values from this column as described below.

Simple 2-to-1 hash

To initiate a 2-to-1 hash of 8 elements ($v_0, ..., v_7$) we need to divide p_0 by the following value:

$$
\alpha_0 + \alpha_1 \cdot m_{bp} + \alpha_2 \cdot r + \sum_{i=0}^7 (\alpha_{i+8} \cdot v_i)
$$

where:
* m_{bp} is a label indicating beginning of a new permutation. Value of this label is computed based on hash chiplet selector flags according to the methodology described [here](../chiplets/hasher.md#multiset-check-constraints).
* r is the address of the row at which the hashing begins.
* Some α values are skipped in the above (e.g., α_3) because of the specifics of how auxiliary hasher table rows are reduced to field elements (described [here](../chiplets/hasher.md#multiset-check-constraints)). For example, α_3 is used as a coefficient for node index values during Merkle path computations in the hasher, and thus, is not relevant in this case. The α_4 term is omitted when the number of items being hashed is a multiple of the rate width (8) because it is multiplied by 0 - the value of the first capacity register as determined by the [hasher chiplet logic](../chiplets/hasher.md#simple-2-to-1-hash).

To read the 4-element result ($u_0, ..., u_3$), we need to divide p_0 by the following value:

$$
\alpha_0 + \alpha_1 \cdot m_{hout} + \alpha_2 \cdot (r + 7) + \sum_{i=0}^3 (\alpha_{i+8} \cdot u_i)
$$

where:
* m_{hout} is a label indicating return of the hash value. Value of this label is computed based on hash chiplet selector flags according to the methodology described [here](../chiplets/hasher.md#multiset-check-constraints).
* r is the address of the row at which the hashing began.

Sequential hash

To initiate a sequential hash of n elements ($v_0, ..., v_{n-1}$), we need to divide p_0 by the following value:

$$
\alpha_0 + \alpha_1 \cdot m_{bp} + \alpha_2 \cdot r + \alpha_4 \cdot n + \sum_{i=0}^7 (\alpha_{i+8} \cdot v_i)
$$

This also absorbs the first 8 elements of the sequence into the hasher state. Then, to absorb the next sequence of 8 elements (e.g., $v_8, ..., v_{15}$), we need to divide p_0 by the following value:

$$
\alpha_0 + \alpha_1 \cdot m_{abp} + \alpha_2 \cdot (r + 7) + \sum_{i=0}^7 (\alpha_{i+8} \cdot v_{i + 8})
$$

Where m_{abp} is a label indicating absorption of more elements into the hasher state. Value of this label is computed based on hash chiplet selector flags according to the methodology described [here](../chiplets/hasher.md#multiset-check-constraints).

We can keep absorbing elements into the hasher in the similar manner until all elements have been absorbed. Then, to read the result (e.g., $u_0, ..., u_3$), we need to divide p_0 by the following value:

$$
\alpha_0 + \alpha_1 \cdot m_{hout} + \alpha_2 \cdot (r + \lceil n / 8 \rceil \cdot 8 - 1) + \sum_{i=0}^3 (\alpha_{i+8} \cdot u_i)
$$

Thus, for example, if $n = 14$, the result will of the hash will be available at hasher row $r + 15$.

Control flow tables

In addition to the hash chiplet, control flow operations rely on 3 virtual tables: *block stack* table, *block hash* table, and _op group_ table. These tables are virtual in that they don't require separate trace columns. Their state is described solely by running product columns: p_1, p_2, and p_3. The tables are described in the following sections.

Block stack table

When the VM starts executing a new program block, it adds its block ID together with the ID of its parent block (and some additional info) to the *block stack* table. When a program block is fully executed, it is removed from the table. In this way, the table represents a stack of blocks which are currently executing on the VM. By the time program execution completes, block stack table must be empty.

The table can be thought of as consisting of 3 columns as shown below:

![decoder_block_stack_table](../../assets/design/decoder/decoder_block_stack_table.png)

where:
* The first column (t_0) contains the ID of the block.
* The second column (t_1) contains the ID of the parent block. If the block has no parent (i.e., it is a root block of the program), parent ID is 0.
* The third column (t_2) contains a binary value which is set to 1 is the block is a *loop* block, and to 0 otherwise.

Running product column p_1 is used to keep track of the state of the table. At any step of the computation, the current value of p_1 defines which rows are present in the table.

To reduce a row in the block stack table to a single value, we compute the following.

$$
row = \alpha_0 + \sum_{i=0}^3 (\alpha_{i+1} \cdot t_i)
$$

Where $\alpha_0, ..., \alpha_3$ are the random values provided by the verifier.

Block hash table

When the VM starts executing a new program block, it adds hashes of the block's children to the *block hash* table. And when the VM finishes executing a block, it removes its hash from the block hash table. Thus, by the time program execution completes, block hash table must be empty.

The table can be thought of as consisting of 7 columns as shown below:

![block_hash_table](../../assets/design/decoder/block_hash_table.png)

where:
* The first column (t_0) contains the ID of the block's parent. For program root, parent ID is 0.
* The next 4 columns ($t_1, ..., t_4$) contain the hash of the block.
* The next column (t_5) contains a binary value which is set to 1 if the block is the first child of a *join* block, and to 0 otherwise.
* The last column (t_6) contains a binary value which is set to 1 if the block is a body of a loop, and to 0 otherwise.

Running product column p_2 is used to keep track of the state of the table. At any step of the computation, the current value of p_2 defines which rows are present in the table.

To reduce a row in the block hash table to a single value, we compute the following.

$$
row = \alpha_0 + \sum_{i=0}^6 (\alpha_{i+1} \cdot t_i)
$$

Where $\alpha_0, ..., \alpha_7$ are the random values provided by the verifier.

Unlike other virtual tables, block hash table does not start out in an empty state. Specifically, it is initialized with a single row containing the hash of the program's root block. This needs to be done because the root block does not have a parent and, thus, otherwise it would never be added to the block hash table.

Initialization of the block hash table is done by setting the initial value of p_2 to the value of the row containing the hash of a program's root block.

Op group table
Op group table is used in decoding of *span* blocks, which are leaves in a program's MAST. As described [here](../programs.md#span-block), a *span* block can contain one or more operation batches, each batch containing up to 8 operation groups.

When the VM starts executing a new batch of operations, it adds all operation groups within a batch, except for the first one, to the *op group* table. Then, as the VM starts executing an operation group, it removes the group from the table. Thus, by the time all operation groups in a batch have been executed, the *op group* table must be empty.

The table can be thought of as consisting of 3 columns as shown below:

![decoder_op_group_table](../../assets/design/decoder/decoder_op_group_table.png)

The meaning of the columns is as follows:

* The first column (t_0) contains operation batch ID. During the execution of the program, each operation batch is assigned a unique ID.
* The second column (t_1) contains the position of the group in the *span* block (not just in the current batch). The position is 1-based and is counted from the end. Thus, for example, if a *span* block consists of a single batch with 4 groups, the position of the first group would be 4, the position of the second group would be 3 etc. (the reason for this is explained in [this](#single-batch-span) section). Note that the group with position 4 is not added to the table, because it is the first group in the batch, so the first row of the table will be for the group with position 3.
* The third column (t_2) contains the actual values of operation groups (this could include up to 9 opcodes or a single immediate value).

Permutation column p_3 is used to keep track of the state of the table. At any step of the computation, the current value of p_3 defines which rows are present in the table.

To reduce a row in the op group table to a single value, we compute the following.

$$
row = \alpha_0 + \sum_{i=0}^2 (\alpha_{i+1} \cdot t_i)
$$

Where $\alpha_0, ..., \alpha_3$ are the random values provided by the verifier.

Control flow operation semantics

In this section we describe high-level semantics of executing all control flow operations. The descriptions are not meant to be complete and omit some low-level details. However, they provide good intuition on how these operations work.

JOIN operation

Before a `JOIN` operation is executed by the VM, the prover populates $h_0, ..., h_7$ registers with hashes of left and right children of the *join* program block as shown in the diagram below.

![decoder_join_operation](../../assets/design/decoder/decoder_join_operation.png)

In the above diagram, `blk` is the ID of the *join* block which is about to be executed. `blk` is also the address of the hasher row in the auxiliary hasher table. `prnt` is the ID of the block's parent.

When the VM executes a `JOIN` operation, it does the following:

1. Adds a tuple `(blk, prnt, 0)` to the block stack table.
2. Adds tuples `(blk, left_child_hash, 1, 0)` and `(blk, right_child_hash, 0, 0)` to the block hash table.
3. Initiates a 2-to-1 hash computation in the hash chiplet (as described [here](#simple-2-to-1-hash)) using `blk` as row address in the auxiliary hashing table and $h_0, ..., h_7$ as input values.

SPLIT operation

Before a `SPLIT` operation is executed by the VM, the prover populates $h_0, ..., h_7$ registers with hashes of true and false branches of the *split* program block as shown in the diagram below.

![decoder_split_operation](../../assets/design/decoder/decoder_split_operation.png)

In the above diagram, `blk` is the ID of the *split* block which is about to be executed. `blk` is also the address of the hasher row in the auxiliary hasher table. `prnt` is the ID of the block's parent.

When the VM executes a `SPLIT` operation, it does the following:

1. Adds a tuple `(blk, prnt, 0)` to the block stack table.
2. Pops the stack and:
   a. If the popped value is 1, adds a tuple `(blk, true_branch_hash, 0, 0)` to the block hash table.
   b. If the popped value is 0, adds a tuple `(blk, false_branch_hash, 0, 0)` to the block hash table.
   c. If the popped value is neither 1 nor 0, the execution fails.
3. Initiates a 2-to-1 hash computation in the hash chiplet (as described [here](#simple-2-to-1-hash)) using `blk` as row address in the auxiliary hashing table and $h_0, ..., h_7$ as input values.

LOOP operation

Before a `LOOP` operation is executed by the VM, the prover populates $h_0, ..., h_3$ registers with hash of the loop's body as shown in the diagram below.

![decoder_loop_operation](../../assets/design/decoder/decoder_loop_operation.png)

In the above diagram, `blk` is the ID of the *loop* block which is about to be executed. `blk` is also the address of the hasher row in the auxiliary hasher table. `prnt` is the ID of the block's parent.

When the VM executes a `LOOP` operation, it does the following:

1. Pops the stack and:
   a. If the popped value is 1 adds a tuple `(blk, prnt, 1)` to the block stack table (the `1` indicates that the loop's body is

expected to be executed). Then, adds a tuple `(blk, loop_body_hash, 0, 1)` to the block hash table.
   b. If the popped value is 0, adds `(blk, prnt, 0)` to the block stack table. In this case, nothing is added to the block hash table.
   c. If the popped value is neither 1 nor 0, the execution fails.
2. Initiates a 2-to-1 hash computation in the hash chiplet (as described [here](#simple-2-to-1-hash)) using `blk` as row address in the auxiliary hashing table and $h_0, ..., h_3$ as input values.

SPAN operation

Before a `SPAN` operation is executed by the VM, the prover populates $h_0, ..., h_7$ registers with contents of the first operation batch of the span block as shown in the diagram below. The prover also sets the group count register gc to the total number of operation groups in the span block.

![decoder_span_block](../../assets/design/decoder/decoder_span_block.png)

In the above diagram, `blk` is the ID of the *span* block which is about to be executed. `blk` is also the address of the hasher row in the auxiliary hasher table. `prnt` is the ID of the block's parent. `g0_op0` is the first operation of the batch, and `g_0'` is the first operation group of the batch with the first operation removed.

When the VM executes a `SPAN` operation, it does the following:

1. Adds a tuple `(blk, prnt, 0)` to the block stack table.
2. Adds groups of the operation batch, as specified by op batch flags (see [here](#operation-batch-flags)) to the op group table.
3. Initiates a sequential hash computation in the hash chiplet (as described [here](#sequential-hash)) using `blk` as row address in the auxiliary hashing table and $h_0, ..., h_7$ as input values.
4. Sets the `in_span` register to 1.
5. Decrements `group_count` register by 1.
6. Sets the `op_index` register to 0.

END operation

Before an `END` operation is executed by the VM, the prover populates $h_0, ..., h_3$ registers with the hash of the block which is about to end. The prover also sets values in h_4 and h_5 registers as follows:
* h_4 is set to 1 if the block is a body of a *loop* block. We denote this value as `f0`.
* h_5 is set to 1 if the block is a *loop* block. We denote this value as `f1`.

![decoder_end_operation](../../assets/design/decoder/decoder_end_operation.png)

In the above diagram, `blk` is the ID of the block which is about to finish executing. `prnt` is the ID of the block's parent.

When the VM executes an `END` operation, it does the following:

1. Removes a tuple `(blk, prnt, f1)` from the block stack table.
2. Removes a tuple `(prnt, current_block_hash, nxt, f0)` from the block hash table, where $nxt=0$ if the next operation is either `END` or `REPEAT`, and 1 otherwise.
3. Reads the hash result from the hash chiplet (as described [here](#program-block-hashing)) using `blk + 7` as row address in the auxiliary hashing table.
4. If $h_5 = 1$ (i.e., we are exiting a *loop* block), pops the value off the top of the stack and verifies that the value is 0.
5. Verifies that `group_count` register is set to 0.

HALT operation

Before a `HALT` operation is executed by the VM, the VM copies values in $h_0, ..., h_3$ registers to the next row as illustrated in the diagram below:

![decoder_halt_operation](../../assets/design/decoder/decoder_halt_operation.png)

In the above diagram, `blk` is the ID of the block which is about to finish executing.

When the VM executes a `HALT` operation, it does the following:

1. Verifies that block address register is set to 0.
2. If we are not at the last row of the trace, verifies that the next operation is `HALT`.
3. Copies values of $h_0, ..., h_3$ registers to the next row.
4. Populates all other decoder registers with 0's in the next row.

REPEAT operation

Before a `REPEAT` operation is executed by the VM, the VM copies values in registers $h_0, ..., h_4$ to the next row as shown in the diagram below.

![decoder_repeat_operation](../../assets/design/decoder/decoder_repeat_operation.png)

In the above diagram, `blk` is the ID of the loop's body and `prnt` is the ID of the loop.

When the VM executes a `REPEAT` operation, it does the following:

1. Checks whether register h_4 is set to 1. If it isn't (i.e., we are not in a loop), the execution fails.

2. Pops the stack and if the popped value is 1, adds a tuple `(prnt, loop_body_loop 0, 1)` to the block hash table. If the popped value is not 1, the execution fails.

The effect of the above is that the VM needs to execute the loop's body again to clear the block hash table.

RESPAN operation

Before a `RESPAN` operation is executed by the VM, the VM copies the ID of the current block `blk` and the number of remaining operation groups in the span to the next row, and sets the value of `in_span` column to 0. The prover also sets the value of h_1 register for the next row to the ID of the current block's parent `prnt` as shown in the diagram below:

![decoder_respan_operation](../../assets/design/decoder/decoder_respan_operation.png)

In the above diagram, `g0_op0` is the first operation of the new operation batch, and `g0'` is the first operation group of the batch with `g0_op0` operation removed.

When the VM executes a `RESPAN` operation, it does the following:

1. Increments block address by 8.
2. Removes the tuple `(blk, prnt, 0)` from the block stack table.
3. Adds the tuple `(blk+8, prnt, 0)` to the block stack table.
4. Absorbs values in registers $h_0, ..., h_7$ into the hasher state of the hash chiplet (as described [here](#sequential-hash)).
5. Sets the `in_span` register to 1.
6. Adds groups of the operation batch, as specified by op batch flags (see [here](#operation-batch-flags)) to the op group table using `blk+8` as batch ID.

The net result of the above is that we incremented the ID of the current block by 8 and added the next set of operation groups to the op group table.

Program decoding

When decoding a program, we start at the root block of the program. We can compute the hash of the root block directly from hashes of its children. The prover provides hashes of the child blocks non-deterministically, and we use them to compute the program's hash (here we rely on the hash chiplet). We then verify the program hash via boundary constraints. Thus, if the prover provided valid hashes for the child blocks, we will get the expected program hash.

Now, we need to verify that the VM executed the child blocks correctly. We do this recursively similar to what is described above: for each of the blocks, the prover provides hashes of its children non-deterministically and we verify that the hash has been computed correctly. We do this until we get to the leaf nodes (i.e., *span* blocks). Hashes of *span* blocks are computed sequentially from the instructions executed by the VM.

The sections below illustrate how different types of code blocks are decoded by the VM.

JOIN block decoding

When decoding a *join* bock, the VM first executes a `JOIN` operation, then executes the first child block, followed by the second child block. Once the children of the *join* block are executed, the VM executes an `END` operation. This is illustrated in the diagram below.

![decoder_join_block_decoding](../../assets/design/decoder/decoder_join_block_decoding.png)

As described previously, when the VM executes a `JOIN` operation, hashes of both children are added to the block hash table. These hashes are removed only when the `END` operations for the child blocks are executed. Thus, until both child blocks are executed, the block hash table is not cleared.

SPLIT block decoding

When decoding a *split* block, the decoder pops an element off the top of the stack, and if the popped element is 1, executes the block corresponding to the `true branch`. If the popped element is 0, the decoder executes the block corresponding to the `false branch`. This is illustrated on the diagram below.

![decoder_split_block_decoding](../../assets/design/decoder/decoder_split_block_decoding.png)

As described previously, when the VM executes a `SPLIT` operation, only the hash of the branch to be executed is added to the block hash table. Thus, until the child block corresponding to the required branch is executed, the block hash table is not cleared.

LOOP block decoding

When decoding a *loop* bock, we need to consider two possible scenarios:

* When the top of the stack is 1, we need to enter the loop and execute loop body at least once.
* When the top of the stack is, 0 we need to skip the loop.

In both cases, we need to pop an element off the top of the stack.

Executing the loop

If the top of the stack is 1, the VM executes a `LOOP` operation. This removes the top element from the stack and adds the hash of the loop's body to the block hash table. It also adds a row to the block stack table setting the `is_loop` value to 1.

To clear the block hash table, the VM needs to execute the loop body (executing the `END` operation for the loop body block will remove the corresponding row from the block hash table). After loop body is executed, if the top of the stack is 1, the VM executes a `REPEAT` operation (executing `REPEAT` operation when the top of the stack is 0 will result in an error). This operation again adds the hash of the loop's body to the block hash table. Thus, the VM needs to execute the loop body again to clear the block hash table.

This process is illustrated on the diagram below.

![decoder_loop_execution](../../assets/design/decoder/decoder_loop_execution.png)

The above steps are repeated until the top of the stack becomes 0, at which point the VM executes the `END` operation. Since in the beginning we set `is_loop` column in the block stack table to 1, h_6 column will be set to 1 when the `END` operation is executed. Thus, executing the `END` operation will also remove the top value from the stack. If the removed value is not 0, the operation will fail. Thus, the VM can exit the loop block only when the top of the stack is 0.

Skipping the loop

If the top of the stack is 0, the VM still executes the `LOOP` operation. But unlike in the case when we need to enter the loop, the VM sets `is_loop` flag to 0 in the block stack table, and does not add any rows to the block hash table. The last point means that the only possible operation to be executed after the `LOOP` operation is the `END` operation. This is illustrated in the diagram below.

![decoder_loop_skipping](../../assets/design/decoder/decoder_loop_skipping.png)

Moreover, since we've set the `is_loop` flag to 0, executing the `END` operation does not remove any items from the stack.

SPAN block decoding

As described [here](../programs.md#span-block), a *span* block can contain one or more operation batches, each batch containing up to 8 operation groups. At the high level, decoding of a span block is done as follows:

1. At the beginning of the block, we make a request to the hash chiplet which initiates the hasher, absorbs the first operation batch (8 field elements) into the hasher, and returns the row address of the hasher, which we use as the unique ID for the *span* block (see [here](#sequential-hash)).
2. We then add groups of the operation batch, as specified by op batch flags (but always skipping the first one) to the op group table.
3. We then remove operation groups from the op group table in the FIFO order one by one, and decode them in the manner similar to the one described [here](#operation-group-decoding).
4. Once all operation groups in a batch have been decoded, we absorb the next batch into the hasher and repeat the process described above.
5. Once all batches have been decoded, we return the hash of the span block from the hasher.

Overall, three control flow operations are used when decoding a *span* block:

1. `SPAN` operation is used to initialize a hasher and absorbs the first operation batch into it.
2. `RESPAN` operation is used to absorb any additional batches in the span block.
3. `END` operation is used to end the decoding of a span block and retrieve its hash from the hash chiplet.

Operation group decoding

As described [here](../programs.md#span-block), an operation group is a sequence of operations which can be encoded into a single field element. For a field element of 64 bits, we can fit up to 9 operations into a group. We do this by concatenating binary representations of opcodes together with the first operation located in the least significant position.

We can read opcodes from the group by simply subtracting them from the op group value and then dividing the result by 2^7. Once the value of the op group reaches 0, we know that all opcodes have been read. Graphically, this can be illustrated like so:

![decoder_operation_group_decoding](../../assets/design/decoder/decoder_operation_group_decoding.png)

Notice that despite their appearance, `op bits` is actually 7 separate registers, while `op group` is just a single register.

We also need to make sure that at most 9 operations are executed as a part of a single group. For this purpose we use the `op_index` column. Values in this column start out at 0 for each operation group, and are incremented by 1 for each executed operation. To make sure that at most 9 operations can be executed in a group, the value of the `op_index` column is not allowed to exceed 8.

Operation batch flags

Operation batch flags are used to specify how many operation groups comprise in a given operation batch. For most batches, the number of groups will be equal to 8. However, for the last batch in a block (or for the first batch, if the block consists of only a single batch), the number of groups may be less than 8. Since processing of new batches starts only on `SPAN` and `RESPAN` operations, only for these operations the flags can be set to non-zero values.

To simplify the constraint system, number of groups in a batch can be only one of the following values: 1, 2, 4, and 8. If number of groups in a batch does not match one of these values, the batch is simply padded with `NOOP`'s (one `NOOP` per added

group). Consider the diagram below.

![decoder_OPERATION_batch_flags](../../assets/design/decoder/decoder_OPERATION_batch_flags.png)

In the above, the batch contains 3 operation groups. To bring the count up to 4, we consider the 4-th group (i.e., 0) to be a part of the batch. Since a numeric value for `NOOP` operation is 0, op group value of 0 can be interpreted as a single `NOOP`.

Operation batch flags (denoted as c_0, c_1, c_2), encode the number of groups and define how many groups are added to the op group table as follows:

* `(1, 0, 0)` - 8 groups. Groups in $h_1, ... h_7$ are added to the op group table.
* `(0, 1, 0)` - 4 groups. Groups in $h_1, ... h_3$ are added to the op group table
* `(0, 0, 1)` - 2 groups. Groups in h_1 is added to the op group table.
* `(0, 1, 1)` - 1 group. Nothing is added to the op group table
* `(0, 0, 0)` - not a `SPAN` or `RESPAN` operation.

Single-batch span

The simplest example of a *span* block is a block with a single batch. This batch may contain up to 8 operation groups (e.g., $g_0, ..., g_7$). Decoding of such a block is illustrated in the diagram below.

![decoder_single_batch_span](../../assets/design/decoder/decoder_single_batch_span.png)

Before the VM starts processing this *span* block, the prover populates registers $h_0, ..., h_7$ with operation groups $g_0, ..., g_7$. The prover also puts the total number of groups into the `group count` register gc. In this case, the total number of groups is 8.

When the VM executes a `SPAN` operation, it does the following:

1. Initiates hashing of elements $g_0, ..., g_7$ using hash chiplet. The hasher address is used as the block ID `blk`, and it is inserted into `addr` register in the next row.
2. Adds a tuple `(blk, prnt, 0)` to the block stack table.
3. Sets the `is_span` register to 1 in the next row.
4. Sets the `op_index` register to 0 in the next row.
5. Decrements `group_count` register by 1.
6. Sets `op bits` registers at the next step to the first operation of g_0, and also copies g_0 with the first operation removed (denoted as g_0') to the next row.
7. Adds groups $g_1, ..., g_7$ to the op group table. Thus, after the `SPAN` operation is executed, op group table looks as shown below.

![decoder_op_group_table_after_span_op](../../assets/design/decoder/decoder_op_group_table_after_span_op.png)

Then, with every step the next operation is removed from g_0, and by step 9, value of g_0 is 0. Once this happens, the VM does the following:

1. Decrements `group_count` register by 1.
2. Sets `op bits` registers at the next step to the first operation of g_1.
3. Sets `hasher` register h_0 to the value of g_1 with the first operation removed (denoted as g_1').
4. Removes row `(blk, 7, g1)` from the op group table. This row can be obtained by taking values from registers: `addr`, `group_count`, and $h_0' + \sum_{i=0}^6(2^i \cdot b_i')$ for $i \in 0..6$, where h_0' and b_i' refer to values in the next row for the first hasher column and `op_bits` columns respectively.

Note that we rely on the `group_count` column to construct the row to be removed from the op group table. Since group count is decremented from the total number of groups to 0, to remove groups from the op group table in correct order, we need to assign group position to groups in the op group table in the reverse order. For example, the first group to be removed should have position 7, the second group to be removed should have position 6 etc.

Decoding of g_1 is performed in the same manner as decoding of g_0: with every subsequent step the next operation is removed from g_1 until its value reaches 0, at which point, decoding of group g_2 begins.

The above steps are executed until value of `group_count` reaches 0. Once `group_count` reaches 0 and the last operation group g_7 is executed, the VM executed the `END` operation. Semantics of the `END` operation are described [here](#end-operation).

Notice that by the time we get to the `END` operation, all rows are removed from the op group table.

Multi-batch span

A *span* block may contain an unlimited number of operation batches. As mentioned previously, to absorb a new batch into the hasher, the VM executes a `RESPAN` operation. The diagram below illustrates decoding of a *span* block consisting of two operation batches.

![decoder_multi_batch_span](../../assets/design/decoder/decoder_multi_batch_span.png)

Decoding of such a block will look very similar to decoding of the single-span block described previously, but there also will be some differences.

First, after the `SPAN` operation is executed, the op group table will look as follows:

```
![decoder_op_group_table_multi_span](../../assets/design/decoder/decoder_op_group_table_multi_span.png)

Notice that while the same groups ($g_1, ..., g_7$) are added to the table, their positions now reflect the total number of groups
in the *span* block.

Second, executing a `RESPAN` operation increments hasher address by $8$. This is done because absorbing additional $8$ elements
into the hasher state requires $8$ more rows in the auxiliary hasher table.

Incrementing value of `addr` register actually changes the ID of the *span* block (though, for a *span* block, it may be more
appropriate to view values in this column as IDs of individual operation batches). This means that we also need to update the
block stack table. Specifically, we need to remove row `(blk, prnt, 0)` from it, and replace it with row `(blk + 8, prnt, 0)`. To
perform this operation, the prover sets the value of $h_1$ in the next row to `prnt`.

Executing a `RESPAN` operation also adds groups $g_9, g_{10}, g_{11}$ to the op group table, which now would look as follows:

![decoder_op_group_table_post_respan](../../assets/design/decoder/decoder_op_group_table_post_respan.png)

Then, the execution of the second batch proceeds in the manner similar to the first batch: we remove operations from the current
op group, execute them, and when the value of the op group reaches $0$, we start executing the next group in the batch. Thus, by
the time we get to the `END` operation, the op group table should be empty.

When executing the `END` operation, the hash of the *span* block will be read from hasher row at address `addr + 7`, which, in our
example, will be equal to `blk + 15`.

#### Handling immediate values

Miden VM operations can carry immediate values. Currently, the only such operation is a `PUSH` operation. Since immediate values
can be thought of as constants embedded into program code, we need to make sure that changing immediate values affects program
hash.

To achieve this, we treat immediate values in a manner similar to how we treat operation groups. Specifically, when computing hash
of a *span* block, immediate values are absorbed into the hasher state in the same way as operation groups are. As mentioned
previously, an immediate value is represented by a single field element, and thus, an immediate value takes place of a single
operation group.

The diagram below illustrates decoding of a *span* block with $9$ operations one of which is a `PUSH` operation.

![decoder_decoding_span_block_with_push](../../assets/design/decoder/decoder_decoding_span_block_with_push.png)

In the above, when the `SPAN` operation is executed, immediate value `imm0` will be added to the op group table, which will look
as follows:

![decoder_imm_vale_op_group_table](../../assets/design/decoder/decoder_imm_vale_op_group_table.png)

Then, when the `PUSH` operation is executed, the VM will do the following:

1. Decrement `group_count` by $1$.
2. Remove a row from the op group table equal to `(addr, group_count, s0')`, where $s_0'$ is the value of the top of the stack at
the next row (i.e., it is the value that is pushed onto the stack).

Thus, after the `PUSH` operation is executed, the op group table is cleared, and group count decreases to $0$ (which means that
there are no more op groups to execute). Decoding of the rest of the op group proceeds as described in the previous sections.

## Program decoding example

Let's run through an example of decoding a simple program shown previously:
```

begin if.true else end end

```
Translating this into code blocks with IDs assigned, we get the following:
```

b0: JOIN b1: SPAN b1: END b2: SPLIT b3: SPAN b3: END b4: SPAN b4: END b2: END b0: END

```
The root of the program is a *join* block $b_0$. This block contains two children: a *span* bock $b_1$ and a *split* block $b_2$.
In turn, the *split* block $b_2$ contains two children: a *span* block $b_3$ and a *span* block $b_4$.

When this program is executed on the VM, the following happens:

1. Before the program starts executing, block hash table is initialized with a single row containing the hash of $b_0$.
2. Then, `JOIN` operation for $b_0$ is executed. It adds hashes of $b_1$ and $b_2$ to the block hash table. It also adds an entry
for $b_0$ to the block stack table. States of both tables after this step are illustrated below.
3. Then, *span* $b_1$ is executed and a sequential hash of its operations is computed. Also, when `SPAN` operation for $b_1$ is
executed, an entry for $b_1$ is added to the block stack table. At the end of $b_1$ (when `END` is executed), entries for $b_1$
are removed from both the block hash and block stack tables.
4. Then, `SPLIT` operation for $b_2$ is executed. It adds an entry for $b_2$ to the block stack table. Also, depending on whether
```

the top of the stack is 1 or 0, either hash of b_3 or hash of b_4 is added to the block hash table. Let's say the top of the stack is 1. Then, at this point, the block hash and block stack tables will look like in the second picture below.
5. Then, *span* b_3 is executed and a sequential hash of its instructions is computed. Also, when `SPAN` operation for b_3 is executed, an entry for b_3 is added to the block stack table. At the end of b_3 (when `END` is executed), entries for b_3 are removed from both the block hash and block stack tables.
6. Then, `END` operation for b_2 is executed. It removes the hash of b_2 from the block hash table, and also removes the entry for b_2 from the block stack table. The third picture below illustrates the states of block stack and block hash tables after this step.
7. Then, `END` for b_0 is executed, which removes entries for b_0 from the block stack and block hash tables. At this point both tables are empty.
8. Finally, a sequence of `HALT` operations is executed until the length of the trace reaches a power of two.

States of block hash and block stack tables after step 2:
![decoder_state_block_hash_2](../../assets/design/decoder/decoder_state_block_hash_2.png)

States of block hash and block stack tables after step 4:
![decoder_state_block_hash_4](../../assets/design/decoder/decoder_state_block_hash_4.png)

States of block hash and block stack tables after step 6:
![decoder_state_block_hash_6](../../assets/design/decoder/decoder_state_block_hash_6.png)
---
id: main
title: Design
sidebar_label: Design
description: "In-depth descriptions of Miden VM internals."
keywords:
  - docs
  - matic
  - polygon
  - wiki
  - miden
  - design
image: https://wiki.polygon.technology/img/thumbnail/polygon-miden.png
---

In the following sections, we provide in-depth descriptions of Miden VM internals, including all AIR constraints for the proving system. We also provide rationale for making specific design choices.

Throughout these sections we adopt the following notations and assumptions:
* All arithmetic operations, unless noted otherwise, are assumed to be in a prime field with modulus $p = 2^{64} - 2^{32} + 1$.
* A _binary_ value means a field element which is either 0 or 1.
* We use lowercase letters to refer to individual field elements (e.g., a), and uppercase letters to refer to groups of 4 elements, also referred to as words (e.g., A). To refer to individual elements within a word, we use numerical subscripts. For example, a_0 is the first element of word A, b_3 is the last element of word B, etc.
* When describing AIR constraints:
  - For a column x, we denote the value in the current row simply as x, and the value in the next row of the column as x'. Thus, all transition constraints for Miden VM work with two consecutive rows of the execution trace.
  - For multiset equality constraints, we denote random values sent by the verifier after the prover commits to the main execution trace as $\alpha_0, \alpha_1, \alpha_2$ etc.
  - To differentiate constraints from other formulas, we frequently use the following format for constraint equations.

$$
x' - (x + y) = 0 \text{ | degree} = 1
$$

In the above, the constraint equation is followed by the implied algebraic degree of the constraint. This degree is determined by the number of multiplications between trace columns. If a constraint does not involve any multiplications between columns, its degree is 1. If a constraint involves multiplication between two columns, its degree is 2. If we need to multiply three columns together, the degree is 3 etc.

The maximum allowed constraint degree in Miden VM is 9. If a constraint degree grows beyond that, we frequently need to introduce additional columns to reduce the degree.

VM components
Miden VM consists of several interconnected components, each providing a specific set of functionality. These components are:

* **Program decoder**, which is responsible for computing a commitment to the executing program and converting the program into a sequence of operations executed by the VM.
* **Operand stack**, which is a push-down stack which provides operands for all operations executed by the VM.
* **Range checker**, which is responsible for providing 16-bit range checks needed by other components.
* **Chiplets**, which is a set of specialized circuits used to accelerate commonly-used complex computations. Currently, the VM relies on 3 chiplets:
  - Hash chiplet, used to compute Rescue Prime Optimized hashes both for sequential hashing and for Merkle tree hashing.
  - Bitwise chiplet, used to compute bitwise operations (e.g., `AND`, `XOR`) over 32-bit integers.
  - Memory chiplet, used to support random-access memory in the VM.

The above components are connected via **buses**, which are implemented using [multiset checks](./multiset.md). We also use multiset checks internally within components to describe **virtual tables**.

VM execution trace
Execution trace of Miden VM consists of 66 main trace columns, 2 buses, and 6 virtual tables as shown in the diagram below.

![vm_trace.png](../assets/design/vm_trace.png)

As can be seen from the above, decoder, stack, and range checker components use dedicated sets of columns, while all chiplets share the same 18 columns. To differentiate between chiplets, we use a set of binary selector columns, a combination of which uniquely identifies each chiplet.

In addition to the components described previously, execution trace also contains 2 system columns:

* `clk` which is used to keep track of the current VM cycle. Values in this column start out at 0 and are incremented by 1 with each cycle.
* `fmp` which contains the value of the free memory pointer used for specifying the region of memory available to procedure locals.

AIR constraints for the `fmp` column are described in [system operations](./stack/system_ops.md) section. For the `clk` column, the constraints are straightforward:

$$
clk' - (clk + 1) = 0 \text{ | degree} = 1
$$
---
id: multiset
title: Multiset Checks
sidebar_label: Multiset Checks
description: "Multiset checks description"
keywords:
  - docs
  - matic
  - wiki
  - polygon
  - miden
  - design
  - chiplet
  - multiset
  - bitwise
image: https://wiki.polygon.technology/img/thumbnail/polygon-miden.png
---

Zero knowledge virtual machines frequently make use of lookup arguments to enable performance optimizations. Miden VM uses multiset checks, and a brief introduction to them can be found [here](https://hackmd.io/@arielg/ByFgSDA7D).

In Miden VM, multiset checks are used for two purposes:

1. To prove the consistency of intermediate values that must persist between different cycles of the trace, without storing the full data in the execution trace (which would require adding more columns to the trace).
2. To prove correct interaction between two independent sections of the execution trace, e.g., between the main trace where the result of some operation is required, but would be expensive to compute, and a specialized component which can perform that operation cheaply.

The first is achieved using [virtual tables](#virtual-tables) of data, where we add a row at some cycle in the trace and remove it at a later cycle when it is needed again. Instead of maintaining the entire table in the execution trace, multiset checks allow us to prove data consistency of this table using one running product column.

The second is done by reducing each operation to a lookup value and then using a [communication bus](#communication-buses), implemented as a running product column, to provably connect the two sections of the trace.

Running product columns
Although the multiset equality check can be thought of as comparing multiset equality between two vectors a and b, in Miden VM it is implemented as a single running product column in the following way:

- The running product column is initialized to a value x at the beginning of the trace. (We typically use $x = 1$.)
- All values of a are multiplied into the running product column.
- All values of b are divided out of the running product column.
- If a and b were multiset equal, then the running product column will equal x at the end of the trace.

Running product columns are computed using a set of random values α_0, $\alpha_1, ...$ sent to the prover by the verifier after the prover commits to the execution trace of the program.

Length of running product columns

Running product columns are computed by including information from the *current* row of the main execution trace into the *next* row of the running product column. Thus, in order to ensure that the trace is long enough to give the running product column space for its final value, a padding row may be required at the end of the trace of the component upon which the running product column depends.

This is true when the data in the main trace could go all the way to the end of the trace, such as in the case of the range checker.

Cost of running product columns
It is important to note that depending on the field in which we operate, a running product column may actually require more than one trace column. This is specifically true for small fields.

Since Miden uses a 64-bit field, each running product column needs to be represented by 2 columns to achieve ~100-bit security and by 3 columns to achieve ~128-bit security.

Virtual tables

Virtual tables can be used to store intermediate data which is computed at one cycle and used at a different cycle. When the data is computed, the row is added to the table, and when it is used later, the row is deleted from the table. Thus, all that needs to be proved is the data consistency between the row that was added and the row that was deleted.

The consistency of a virtual table can be proved with a single trace column p, which keeps a running product of rows that were inserted into and deleted from the table. This is done by reducing each row to a single value, multiplying the value into p when the row is inserted, and dividing the value out of p when the row is removed. Thus, at any step of the computation, p will contain a product of all rows currently in the table.

The initial value of p is set to 1. Thus, if the table is empty by the time Miden VM finishes executing a program (we added and then removed exactly the same set of rows), the final value of p will also be equal to 1. The initial and final values are enforced via boundary constraints.

Computing a virtual table's trace column

To compute a product of rows, we'll first need to reduce each row to a single value. This can be done as follows.

Let $t_0, t_1, t_2, ...$ be columns in the virtual table, and assume the verifier sends a set of random values α_0, $\alpha_1, ...$ to the prover after the prover commits to the execution trace of the program.

The prover reduces row i in the table to a single value r_i as:

$$
r_i = \alpha_0 + \alpha_1 \cdot t_{0, i} + \alpha_2 \cdot t_{1, i} + \alpha_3 \cdot t_{2, i} + ...
$$

Then, when row i is added to the table, we'll update the value in the p column like so:

$$
p' = p \cdot r_i
$$

Analogously, when row i is removed from the table, we'll update the value in column p like so:

$$
p' = \frac{p}{r_i}
$$

Virtual tables in Miden VM

Miden VM currently makes use of 6 virtual tables across 4 components:

- Stack:
    - [Overflow table](./stack/main.md#overflow-table)
- Decoder:
    - [Block stack table](./decoder/main.md#block-stack-table)
    - [Block hash table](./decoder/main.md#block-hash-table)
    - [Op group table](./decoder/main.md#op-group-table)
- Range checker:
    - [8-bit range checks table](./range.md#8-bit-range-checks-table)
- Hash chiplet:
    - [Sibling table](./chiplets/hasher.md#sibling-table-constraints)

Communication buses

One strategy for improving the efficiency of a zero knowledge virtual machine is to use specialized components for complex operations and have the main circuit “offload” those operations to the corresponding components by specifying inputs and outputs and allowing the proof of execution to be done by the dedicated component instead of by the main circuit.

These specialized components are designed to prove the internal correctness of the execution of the operations they support. However, in isolation they cannot make any guarantees about the source of the input data or the destination of the output data.

In order to prove that the inputs and outputs specified by the main circuit match the inputs and outputs provably executed in the specialized component, some kind of provable communication bus is needed.

This bus is typically implemented as some kind of lookup argument, and in Miden VM in particular we use multiset checks.

Implementation

A `bus` can be implemented as a single trace column b where a request can be sent to a specific component and a corresponding response will be sent back by that component.

The values in this column contain a running product of the communication with the component as follows:

- Each request is “sent” by computing a lookup value from some information that's specific to the specialized component, the operation inputs, and the operation outputs, and then dividing it out of the running product column b.
- Each chiplet response is “sent” by computing the same lookup value from the component-specific information, inputs, and outputs, and then multiplying it into the running product column b.

Thus, if the requests and responses match, and the bus column b is initialized to 1, then b will start and end with the value 1. This condition is enforced by boundary constraints on column b.

Note that the order of the requests and responses does not matter, as long as they are all included in b. In fact, requests and responses for the same operation will generally occur at different cycles. Additionally, there could be multiple requests sent in the same cycle, and there could also be a response provided at the same cycle that a request is received.

Communication bus constraints

These constraints can be expressed in a general way with the 2 following requirements:

- The lookup value must be computed using random values α_0, α_1, etc. that are provided by the verifier after the prover has committed to the main execution trace.
- The lookup value must include all uniquely identifying information for the component/operation and its inputs and outputs.

Given an example operation op_{ex} with inputs $i_0, ..., i_n$ and outputs $o_0, ..., o_m$, the lookup value can be computed as follows:

$$lookup = \alpha_0 + \alpha_1 \cdot op_{ex} + \alpha_2 \cdot i_0 + ... + \alpha_{n+2} \cdot i_n + \alpha_{n+3} \cdot o_0 + ... + \alpha_{n + 2 + m} \cdot o_m$$

The constraint for sending this to the bus as a request would be:

$$b' \cdot lookup = b$$

The constraint for sending this to the bus as a response would be:

$$b' = b \cdot lookup$$

However, these constraints must be combined, since it's possible that requests and responses both occur during the same cycle.

To combine them, let u_{lookup} be the request value and let v_{lookup} be the response value. These values are both computed the same way as shown above, but the data sources are different, since the input/output values used to compute u_{lookup} come from the trace of the component that's "offloading" the computation, while the input/output values used to compute v_{lookup} come from the trace of the specialized component.

The final constraint can be expressed as:

$$b' \cdot u_{lookup} = b \cdot v_{lookup}$$

Communication buses in Miden VM

In Miden VM, the specialized components are implemented as dedicated segments of the execution trace, which include the range checker and the 3 chiplets in the Chiplets module (the hash chiplet, bitwise chiplet, and memory chiplet).

Miden VM currently uses 2 buses to communicate with these components:

- The chiplets bus [b_{chip}](./chiplets/main.md#chiplet-bus), which communicates with all of the chiplets (Hash, Bitwise, and Memory).
- The range checker bus [b_{range}](./range.md#communication-bus).

---
id: programs
title: Programs in Miden VM
sidebar_label: Programs
description: "How programs are executed in Miden VM."
keywords:
  - docs
  - matic
  - polygon
  - wiki
  - miden
  - design
  - code
  - block
  - computation

image: https://wiki.polygon.technology/img/thumbnail/polygon-miden.png
---

Miden VM consumes programs in a form of a Merkelized Abstract Syntax Tree (MAST). This tree is a binary tree where each node is a *code block*. The VM starts execution at the root of the tree, and attempts to recursively execute each required block according to its semantics. If the execution of a code block fails, the VM halts at that point and no further blocks are executed. A set of currently available blocks and their execution semantics are described below.

Code blocks

Join block
A **join** block is used to describe sequential execution. When the VM encounters a *join* block, it executes its left child first, and then executes its right child.

![join_block](../assets/design/programs/join_block.png)

A *join* block must always have two children, and thus, cannot be a leaf node in the tree.

Split block
A **split** block is used to describe conditional execution. When the VM encounters a *split* block, it checks the top of the stack. If the top of the stack is 1, it executes the left child, if the top of the stack is 0, it executes the right child. If the top of the stack is neither 0 nor 1, the execution fails.

![split_block](../assets/design/programs/split_block.png)

A *split* block must always have two children, and thus, cannot be a leaf node in the tree.

Loop block
A **loop** block is used to describe condition-based iterative execution. When the VM encounters a *loop* block, it checks the top of the stack. If the top of the stack is 1, it executes the loop body, if the top of the stack is 0, the block is not executed. If the top of the stack is neither 0 nor 1, the execution fails.

After the body of the loop is executed, the VM checks the top of the stack again. If the top of the stack is 1, the body is executed again, if the top of the stack is 0, the loop is exited. If the top of the stack is neither 0 nor 1, the execution fails.

![loop_block](../assets/design/programs/loop_block.png)

A *loop* block must always have one child, and thus, cannot be a leaf node in the tree.

Call block

A **call** block is used to describe a function call which is executed in a [user context](../user_docs/assembly/execution_contexts.md). When the VM encounters a *call* block, it creates a new user context, then executes a program which hashes to the target specified by the *call* block in the new context. Thus, in order to execute a *call* block, the VM must be aware of a program with the specified hash. Otherwise, the execution fails. At the end of the *call* block, execution returns to the previous context.

When executing a *call* block, the VM does the following:
1. Checks if a *syscall* is already being executed and fails if so.
2. Sets the depth of the stack to 16.
3. Upon return, checks that the depth of the stack is 16. If so, the original stack depth is restored. Otherwise, an error occurs.

![call_block](../assets/design/programs/call_block.png)

A *call* block does not have any children. Thus, it must be leaf node in the tree.

Syscall block

A **syscall** block is used to describe a function call which is executed in the [root context](../user_docs/assembly/execution_contexts.md). When the VM encounters a *syscall* block, it returns to the root context, then executes a program which hashes to the target specified by the *syscall* block. Thus, in order to execute a *syscall* block, the VM must be aware of a program with the specified hash, and that program must belong to the kernel against which the code is compiled. Otherwise, the execution fails. At the end of the *syscall* block, execution returns to the previous context.

When executing a *syscall* block, the VM does the following:
1. Checks if a *syscall* is already being executed and fails if so.
2. Sets the depth of the stack to 16.
3. Upon return, checks that the depth of the stack is 16. If so, the original stack depth is restored. Otherwise, an error occurs.

![syscall_block](../assets/design/programs/syscall_block.png)

A *syscall* block does not have any children. Thus, it must be leaf node in the tree.

Span block
A **span** block is used to describe a linear sequence of operations. When the VM encounters a *span* block, it breaks the sequence of operations into batches and groups according to the following rules:

```
* A group is represented by a single field element. Thus, assuming a single operation can be encoded using 7 bits, and assuming we
are using a 64-bit field, a single group may encode up to 9 operations or a single immediate value.
* A batch is a set of groups which can be absorbed by a hash function used by the VM in a single permutation. For example,
assuming the hash function can absorb up to 8 field elements in a single permutation, a single batch may contain up to 8 groups.
* There is no limit on the number of batches contained within a single span.

Thus, for example, executing 8 pushes in a row will result in two operation batches as illustrated in the picture below:

![span_block_creation](../assets/design/programs/span_block_creation.png)

* The first batch will contain 8 groups, with the first group containing 7 `PUSH` opcodes and 1 `NOOP`, and the remaining 7 groups
containing immediate values for each of the push operations. The reason for the `NOOP` is explained later in this section.
* The second batch will contain 2 groups, with the first group containing 1 `PUSH` opcode and 1 `NOOP`, and the second group
containing the immediate value for the last push operation.


If a sequence of operations does not have any operations which carry immediate values, up to 72 operations can fit into a single
batch.

From the user's perspective, all operations are executed in order, however, the VM may insert occasional `NOOP`s to ensure proper
alignment of all operations in the sequence. Currently, the alignment requirements are as follows:
* An operation carrying an immediate value cannot be the last operation in a group. Thus, for example, if a `PUSH` operation is
the last operation in a group, the VM will insert a `NOOP` after it.

A *span* block does not have any children, and thus, must be leaf node in the tree.

## Program example
Consider the following program, where $a_0, ..., a_i$, $b_0, ..., b_j$ etc. represent individual operations:
```

a_0, ..., a_i if.true b_0, ..., b_j else c_0, ..., c_k while.true d_0, ..., d_n end e_0, ..., e_m end f_0, ..., f_l

```

A MAST for this program would look as follows:

![mast_of_program](../assets/design/programs/mast_of_program.png)

Execution of this program would proceed as follows:

1. The VM will start execution at the root of the program which is block $B_5$.
2. Since, $B_5$ is a *join block*, the VM will attempt to execute block $B_4$ first, and only after that execute block $f$.
3. Block $B_4$ is also a *join block*, and thus, the VM will execute block $a$ by executing operations $a_0, ..., a_i$ in
sequence, and then execute block $B_3$.
4. Block $B_3$ is a *split block*, and thus, the VM will pop the value off the top of the stack. If the popped value is $1$,
operations from block $b$ will be executed in sequence. If the popped value is $0$, then the VM will attempt to execute block
$B_2$.
5. $B_2$ is a *join block*, thus, the VM will try to execute block $B_1$ first, and then execute operations from block $e$.
6. Block $B_1$ is also a *join_block*, and thus, the VM will first execute all operations in block $c$, and then will attempt to
execute block $B_0$.
7. Block $B_0$ is a loop block, thus, the VM will pop the value off the top of the stack. If the pooped value is $1$, the VM will
execute the body of the loop defined by block $d$. If the popped value is $0$, the VM will not execute block $d$ and instead will
move up the tree executing first block $e$, then $f$.
8. If the VM does enter the loop, then after operation $d_n$ is executed, the VM will pop the value off the top of the stack
again. If the popped value is $1$, the VM will execute block $d$ again, and again until the top of the stack becomes $0$. Once the
top of the stack becomes $0$, the VM will exit the loop and will move up the tree executing first block $e$, then $f$.

## Program hash computation
Every Miden VM program can be reduced to a unique hash value. Specifically, it is infeasible to find two Miden VM programs with
distinct semantics which hash to the same value. Padding a program with `NOOP`s does not change a program's execution semantics,
and thus, programs which differ only in the number and/or placement of `NOOP`s may hash to the same value, although in most cases
padding with `NOOP` should not affect program hash.

To prevent program hash collisions we implement domain separation across the variants of control blocks. We define the domain
value to be the opcode of the operation that initializes the control block.

Below we denote $hash$ to be an arithmetization-friendly hash function with $4$-element output and capable of absorbing $8$
elements in a single permutation. The hash domain is specified as the subscript of the hash function and its value is used to
populate the second capacity register upon initialization of control block hashing - $hash_{domain}(a, b)$.

* The hash of a **join** block is computed as $hash_{join}(a, b)$, where $a$ and $b$ are hashes of the code block being joined.
* The hash of a **split** block is computed as $hash_{split}(a, b)$, where $a$ is a hash of a code block corresponding to the
*true* branch of execution, and $b$ is a hash of a code block corresponding to the *false branch* of execution.
* The hash of a **loop** block is computed as $hash_{loop}(a, 0)$, where $a$ is a hash of a code block corresponding to the loop
body.
* The hash of a **call** block is computed as $hash_{call}(a, 0)$, where $a$ is a hash of a program of which the VM is aware.
* The hash of a **syscall** block is computed as $hash_{syscall}(a, 0)$, where $a$ is a hash of a program belonging to the kernel
against which the code was compiled.
* The hash of a **span** block is computed as $hash(a_1, ..., a_k)$, where $a_i$ is the $i$th batch of operations in the *span*
block. Each batch of operations is defined as containing $8$ field elements, and thus, hashing a $k$-batch *span* block requires
```

k absorption steps.
    * In cases when the number of operations is insufficient to fill the last batch entirely, `NOOPs` are appended to the end of the last batch to ensure that the number of operations in the batch is always equal to 8.
---
id: range
title: Range Checker
sidebar_label: Range Checker
description: "How 16-bit range checks are performed."
keywords:
  - docs
  - matic
  - polygon
  - wiki
  - miden
  - design
  - range
image: https://wiki.polygon.technology/img/thumbnail/polygon-miden.png
---

Miden VM relies very heavily on 16-bit range-checks (checking if a value of a field element is between 0 and 2^{16}). For example, most of the [u32 operations](./stack/u32_ops.md), need to perform between two and four 16-bit range-checks per operation. Similarly, operations involving memory (e.g. load and store) require two 16-bit range-check per operation.

Thus, it is very important for the VM to be able to perform a large number 16-bit range checks very efficiently. In this note we describe how this can be achieved using permutation checks.

8-bit range checks

First, let's define a construction for the simplest possible 8-bit range-check. This can be done with a single column as illustrated below.

![rc_8_bit_range_check](../assets/design/range/rc_8_bit_range_check.png)

For this to work as a range-check we need to enforce a few constraints on this column:

- Value in the first row must be 0.
- Value in the last row must be 255.
- As we move from one row to the next, we can either keep the value the same, or increment it by 1.

Denoting v as the value of column v in the current row, and v' as the value of column v in the next row, we can enforce the last condition as follows:

$$
(v' - v) \cdot (v' - v - 1) = 0
$$

Together, these constraints guarantee that all values in column v are between 0 and 255 (inclusive).

We can then add another column p_0, which will keep a running product of values in v offset by random value α_0 (provided by the verifier). Transition constraints for column p_0 would look like so:

$$
p'_0 - p_0 \cdot (\alpha_0 + v) = 0
$$

Using these two columns we can check if some other column in the execution trace is a permutation of values in v. Let's call this other column x. We can compute the running product of x in the same way as we compute the running product for v. Then, we can check that the last value in p_0 is the same as the final value for the running product of x (this is the permutation check).

While this approach works, it has a couple of limitations:

- First, column v must contain all values between 0 and 255. Thus, if column x does not contain one of these values, we need to artificially add this value to x somehow (i.e., we need to pad x with extra values).
- Second, assuming n is the length of execution trace, we can range-check at most n values. Thus, if we wanted to range-check more than n values, we'd need to introduce another column similar to v.

To get rid of the padding requirement, we can add a _selector_ column, which would contain 1 for values we want to include in the running product, and 0 for the ones we don't. But we can address both issues with a single solution.

A better construction

Let's add two selector column to our table s_0 and s_1 as illustrated below.

![rc_better_construction](../assets/design/range/rc_better_construction.png)

The purpose of these columns is as follows:

- When $s_0 = 0$ and $s_1 = 0$, we won't include the value into the running product.
- When $s_0 = 1$ and $s_1 = 0$, we will include the value into the running product.
- When $s_0 = 0$ and $s_1 = 1$, we will include two copies of the value into the running product.
- When $s_0 = 1$ and $s_1 = 1$, we will include four copies of the value into the running product.

Thus, for the table pictured below, the running product will include: a single 0, a single 1, no 2's, and five 3's etc.

To keep the description of constraints simple, we'll first define the four flag values as follows:

$$
f_0 = (1 - s_0) \cdot (1 - s_1)
$$

$$
f_1 = s_0 \cdot (1 - s_1)
$$

$$
f_2 = (1 - s_0) \cdot s_1
$$

$$
f_3 = s_0 \cdot s_1
$$

Thus, for example, when $s_0 = 1$ and $s_1 = 1$, $f_3 = 1$ and $f_0 = f_1 = f_2 = 0$.

Then, we'll update transition constraints for p_0 like so:

$$
p'_0 - p_0 \cdot \left((\alpha_0 + v)^4 \cdot f_3 + (\alpha_0 + v)^2 \cdot f_2 + (\alpha_0 + v) \cdot f_1 + f_0\right) = 0
$$

The above ensures that when $f_0 = 1$, p_0 remains the same, when $f_1 = 1$, p_0 is multiplied by $(\alpha_0 + v)$, when $f_2 = 1$, p_0 is multiplied by $(\alpha_0 + v)^2$, and when $f_3 = 1$, p_0 is multiplied by $(\alpha_0 + v)^4$.

We also need to ensure that values in columns s_0 and s_1 are binary (either 0 or 1). This can be done with the following constraints:

$$
s_0^2 - s_0 = 0
$$

$$
s_1^2 - s_1 = 0
$$

And lastly, for completeness, we still need to impose a transition constraint that we had in the naive approach:

$$
(v' - v) \cdot (v' - v - 1) = 0
$$

This 3-column table addresses the limitations we had as follows:

1. We no longer need to pad the column we want to range-check with extra values because we can skip the values we don't care about.
2. We can support almost $4n$ range checks (when n is relatively large). Though, for short traces (when $n < 256$), we can range-check at most n unique values.

The one downside of this approach is that the degree of our constraints is now 6 (vs. 2 in the naive approach). But in the context of Miden VM this doesn't matter as maximum constraint degree for the VM is 8 anyway.

16-bit range checks

To support 16-bit range checks, let's try to extend the idea of the 8-bit table. Our 16-bit table would look like so (the only difference is that column u now has to end with value 65535):

![rc_16_bit_range_check](../assets/design/range/rc_16_bit_range_check.png)

While this works, it is rather wasteful. In the worst case, we'd need to enumerate over 65K values, most of which we may not actually need. It would be nice if we could "skip over" the values that we don't want. We can do this by relying on 8-bit range checks. Specifically, instead of enforcing constraint:

$$
(u' - u) \cdot (u' - u - 1) = 0
$$

We would enforce:

$$
p'_1 - p_1 \cdot (\alpha_0 + u' - u) = 0
$$

Where p_1 is another running product column. At the end of the execution trace, we would check that $p_0 = p_1$. This would ensure that as we move from one row to another, values in column u increase by at most 255 (we are basically performing an 8-bit range check on increments of column u). Now, our table can look like this:

![rc_table_post_8_bit_range_check](../assets/design/range/rc_table_post_8_bit_range_check.png)

We still may need to include some unneeded rows because we can not "jump" by more than 255 values, but at least we are guaranteed that the number of such unneeded rows will never be greater than 256.

We also need to add another running product column p_2 to support permutation checks against column u. The constraint that we'll need to impose against this column is identical to the constraint we imposed against column p_0 and will look like so:

$$
p'_2 - p_2 \cdot \left((\alpha_0 + u)^4 \cdot f_3 + (\alpha_0 + u)^2 \cdot f_2 + (\alpha_0 + u) \cdot f_1 + f_0\right) = 0
$$

Overall, with this construction we have the following:

- We need two table of three columns each (6 columns total), and we need 3 running product columns.
- This gives us the ability to do the following:
  - For long traces (when $n > 2^{16}$) we can support almost $4n$ arbitrary 16-bit range-checks.
  - For short traces, we can range-check at most n unique values, but if there are duplicates, we can support up to $4n$ total range-checks.

But we can do better.

Optimizations

First, we can just stack the tables on top of each other. We'll need to add a column to partition the table between the sections used for 8-bit range checks and sections used for 16-bit range checks. Let's call this column t. When $t = 0$, we'll apply constraints for the 8-bit table, and when $t = 1$, we'll apply constraints for the 16-bit table.

![rc_optimisation.png](../assets/design/range/rc_optimisation.png)

Second, we can merge running product columns p_0 and p_1 into a single column. We'll do it like so:

- When $t = 0$, we'll multiply the current value of the column by the 8-bit value (offset by random α_0) that we want to add into the running product.
- When $t = 1$, we'll divide the current value of the column by the 8-bit value (offset by random α_0) which we'd like to remove from the running product.

In the end, if we added and then removed all the same values, the value in this column (let's call it p_0) should equal to 1, and we can check this condition via a boundary constraint.

The only downside of this construction is again higher constraint degree. Specifically, some of the transition constraints described above have degree 8. However, this doesn't matter in the context of Miden VM, since the max constraint degree of the VM is 9.

Miden approach

This final optimized construction is implemented in Miden with the following requirements, capabilities, and constraints.

Requirements

- 4 columns of the main trace: t, s_0, s_1, v.
- 1 [bus](./multiset.md#communication-buses) b_{range} to ensure that the range checks performed in the range checker match those requested by other VM components (the [stack](./stack/u32_ops.md#range-checks) and the [memory chiplet](./chiplets/memory.md)).
- 1 [virtual table](./multiset.md#virtual-tables), tracked in running product column p_0, which enables skipping values by up to 255 in the 16-bit section of the trace and enforces consistency between the range checks in the 8-bit section of the trace and the value increments in the 16-bit section of the trace.

Capabilities

The construction gives us the following capabilities:
- For long traces (when $n > 2^{16}$), we can do over $3n$ arbitrary 16-bit range-checks.
- For short traces ($2^{10} < n \le 2^{16}$), we can range-check at slightly fewer than n unique values, but if there are duplicates, we may be able to range-check up to $3n$ total values.

Execution trace

The range checker's execution trace looks like the table described in the [optimizations](#optimizations) section above.

![rc_optimisation.png](../assets/design/range/rc_optimisation.png)

As previously described, the columns have the following meanings:
- t contains a binary value which differentiates between the 8-bit ($t = 0$) and 16-bit ($t = 1$) sections of the range checker's trace.
  - The 8-bit section of the trace contains the range checks required to ensure internal consistency of the Range Checker.
  - The 16-bit section of the trace contains the range checks required by other components (e.g. the stack and the memory chiplet).
- s_0 and s_1 are selector columns that are combined into flags to indicate the number of times the value in that row should be range checked (included into the running product). With these flags, values can be included 0, 1, 2, or 4 times per row in the execution trace. (Values can be included more times by having multiple trace rows with the same value).
- v contains the values to be range checked.
  - During the 8-bit section of the trace (when $t = 0$), these values go from 0 to 255 and must either stay the same or increase by one at each step.
  - During the 16-bit section of the trace (when $t = 1$), these values go from 0 to 65535. Values must either stay the same or increase by less than 256 at each step.
  - The final 2 rows of the 16-bit section of the trace must both equal 65535. The extra value of 65535 is required in order to [pad the trace](./multiset.md#length-of-running-product-columns) so the [b_{range}](#communication-bus) running product bus column can be computed correctly.

Execution trace constraints

First, we'll need to make sure that all selector flags are binary. This can be done with the following constraints:

$$
t^2 - t = 0 \text{ | degree} = 2
$$

$$
s_0^2 - s_0 = 0 \text{ | degree} = 2
$$

$$
s_1^2 - s_1 = 0 \text{ | degree} = 2
$$

Next, we need to constrain the row transitions in the 8-bit section of the table so that as we move from one row to the next the value either stays the same or increases by 1.

$$
(1 - t') \cdot (v' - v) \cdot (v' - v - 1) = 0 \text{ | degree} = 3
$$

Next, we need to make sure that values in column t can "flip" from 0 to 1 only once. The following constraint enforces this:

$$
t \cdot (1 - t') = 0 \text{ | degree} = 2
$$

Finally, we need to make sure that when column t "flips" from 0 to 1 (we are moving from the 8-bit section of the table to the 16-bit section), the current value in column v is equal to 255, and the next value is reset to 0. This can be done with the following constraints:

$$
(1 - t) \cdot t' \cdot (v - 255) = 0 \text{ | degree} = 3
$$

$$
(1 - t) \cdot t' \cdot v' = 0 \text{ | degree} = 3
$$

In addition to the transition constraints described above, we also need to enforce the following boundary constraints:

- Value of v in the first row is 0.
- Value of v in the last row is 65535.

8-bit range checks table

The 8-bit range checks [virtual table](./multiset.md#virtual-tables) is used to enforce the internal correctness of the 16-bit section of the Range Checker (where range checks for user operations and other components are executed).

This table can be thought of as a virtual table that contains all 8-bit range checks required to ensure correctness of the 16-bit section:

- When a value is range-checked in the 8-bit section of the trace (i.e., the flag indicates that it should be included 1, 2, or 4 times), it is added to the virtual table. The value must be added to the virtual table as many times as that row's flag indicates it should be range-checked.

- In the 16-bit section of the trace, a single 8-bit range check on $v' - v$ is required at each step. Therefore, a single 8-bit range check of $v' - v$ is removed from the virtual table at each step.

The running product column p_0 is used to keep track of the state of the table.

To simplify the notation, we'll first define variable z, which represents how a row in the execution trace is reduced to a single value.

$$
z = (\alpha_0 + v)^4 \cdot f_3 + (\alpha_0 + v)^2 \cdot f_2 + (\alpha_0 + v) \cdot f_1 + f_0
$$

In the 8-bit section ($t = 0$), one, two, or four 8-bit range checks are added to the virtual table at each step and p_0 is updated as follows:

$$
p'_0 = p_0 \cdot z
$$

Note that if f_0 is true then the value of p_0 does not change, so no 8-bit range checks are added to the virtual table.

In the 16-bit section ($t = 1$), one 8-bit range check is removed from the virtual table at each step and p_0 is updated as follows:

$$
p'_0 \cdot (\alpha_0 + v' - v) = p_0
$$

The above actually enforces that $p'_0 = p_0 / (\alpha_0 + v' - v)$.

These two updates, which first build up the product and then reduce it, can be combined into a single constraint:

$$
p'_0 \cdot ((\alpha_0 + v' - v) \cdot t - t + 1) = p_0 \cdot (z - z \cdot t + t) \text{ | degree} = 8
$$

Thus, if the prover arranged the 8-bit and the 16-bit sections of the table correctly and p_0 was initialized to 1, we should end up with $p_0 = 1$ at the end of the trace.

In addition to the transition constraints described above, we also need to enforce the following boundary constraint:

- The value of p_0 in the first and last rows is 1.

Communication bus

b_{range} is the [bus](./multiset.md#communication-buses) that connects components which require 16-bit range checks to the range-checked values in the 16-bit section of the range checker. The bus constraints are defined by the components that use it to communicate.

Requests are sent to the range checker bus by the following components:
- The Stack sends requests for 16-bit range checks during some [`u32` operations](./stack/u32_ops.md#range-checks).
- The [Memory chiplet](./chiplets/memory.md) sends requests for 16-bit range checks against the values in the d_0 and d_1 trace columns to enforce internal consistency.

Responses are provided by the range checker as follows.

Once again, we'll make use of variable z, which represents how a row in the execution trace is reduced to a single value.

$$
z = (\alpha_0 + v)^4 \cdot f_3 + (\alpha_0 + v)^2 \cdot f_2 + (\alpha_0 + v) \cdot f_1 + f_0
$$

Only the 16-bit section of the trace should be included in the b_{range} bus column. Transition constraints for this are fairly straightforward:

$$
b'_{range} = b_{range} \cdot (z \cdot t - t + 1) \text{ | degree} = 8
$$

Thus, when $t = 0$, the value in b_{range} does not change, but when $t = 1$, the next value in b_{range} is computed by multiplying the current value by z.

If b_{range} is initialized to 1 and the values sent to the bus by other VM components match those that are range-checked in the 16-bit section of the trace, then at the end of the trace we should end up with $b_{range} = 1$.

In addition to the transition constraint described above, we also need to enforce the following boundary constraint:

- The value of b_{range} in the first and last rows is 1.

---
id: crypto_ops
title: Cryptographic Operations
sidebar_label: Cryptographic Operations
description: "AIR constraints for Miden VM cryptographic operations."
keywords:
  - docs
  - matic
  - polygon
  - wiki
  - miden
  - design
  - cryptography
image: https://wiki.polygon.technology/img/thumbnail/polygon-miden.png
---

In this section we describe the AIR constraints for Miden VM cryptographic operations.

Cryptographic operations in Miden VM are performed by the [Hash chiplet](../chiplets/hasher.md). Communication between the stack and the hash chiplet is accomplished via the chiplet bus b_{chip}. To make requests to and to read results from the chiplet bus we need to divide its current value by the value representing the request.

Thus, to describe AIR constraints for the cryptographic operations, we need to define how to compute these input and output values within the stack. We do this in the following sections.

HPERM
The `HPERM` operation applies Rescue Prime Optimized permutation to the top 12 elements of the stack. The stack is assumed to be arranged so that the 8 elements of the rate are at the top of the stack. The capacity word follows, with the number of elements to be hashed at the deepest position in stack. The diagram below illustrates this graphically.

![hperm](../../assets/design/stack/crypto_ops/HPERM.png)

In the above, r (located in the helper register h_0) is the row address from the hash chiplet set by the prover non-deterministically.

For the `HPERM` operation, we define input and output values as follows:

$$
v_{input} = \alpha_0 + \alpha_1 \cdot op_{linhash} + \alpha_2 \cdot h_0 + \sum_{j=0}^{11} (\alpha_{j+4} \cdot s_{11-i})
$$

$$
v_{output} = \alpha_0 + \alpha_1 \cdot op_{retstate} + \alpha_2 \cdot (h_0 + 7) + \sum_{j=0}^{11} (\alpha_{i+4} \cdot s_{11-i}')
$$

In the above, $op_{linhash}$ and $op_{retstate}$ are the unique [operation labels](../chiplets/main.md#operation-labels) for initiating a linear hash and reading the full state of the hasher respectively. Also note that the term for α_3 is missing from the above expressions because for Rescue Prime Optimized permutation computation the index column is expected to be set to 0.

Using the above values, we can describe the constraint for the chiplet bus column as follows:

$$
b_{chip}' \cdot v_{input} \cdot v_{output} = b_{chip} \text{ | degree} = 3
$$

The above constraint enforces that the specified input and output rows must be present in the trace of the hash chiplet, and that they must be exactly 7 rows apart.

The effect of this operation on the rest of the stack is:
* **No change** starting from position 12.

MPVERIFY
The `MPVERIFY` operation verifies that a Merkle path from the specified node resolves to the specified root. This operation can be used to prove that the prover knows a path in the specified Merkle tree which starts with the specified node.

Prior to the operation, the stack is expected to be arranged as follows (from the top):
- Value of the node, 4 elements (V in the below image)
- Depth of the path, 1 element (d in the below image)
- Index of the node, 1 element (i in the below image)
- Root of the tree, 4 elements (R in the below image)

The Merkle path itself is expected to be provided by the prover non-deterministically (via the advice provider). If the prover is not able to provide the required path, the operation fails. Otherwise, the state of the stack does not change. The diagram below illustrates this graphically.

![mpverify](../../assets/design/stack/crypto_ops/MPVERIFY.png)

In the above, r (located in the helper register h_0) is the row address from the hash chiplet set by the prover non-deterministically.

For the `MPVERIFY` operation, we define input and output values as follows:

$$
v_{input} = \alpha_0 + \alpha_1 \cdot op_{mpver} + \alpha_2 \cdot h_0 + \alpha_3 \cdot s_5 + \sum_{j=0}^3 \alpha_{j+8} \cdot s_{3 - j}
$$

$$
v_{output} = \alpha_0 + \alpha_1 \cdot op_{rethash} + \alpha_2 \cdot (h_0 + 8 \cdot s_4 - 1) + \sum_{j=0}^3\alpha_{j + 8} \cdot s_{9 - j}
$$

In the above, op_{mpver} and $op_{rethash}$ are the unique [operation labels](../chiplets/main.md#operation-labels) for initiating a Merkle path verification computation and reading the hash result respectively. The sum expression for inputs computes the value of the leaf node, while the sum expression for the output computes the value of the tree root.

Using the above values, we can describe the constraint for the chiplet bus column as follows:

$$
b_{chip}' \cdot v_{input} \cdot v_{output} = b_{chip} \text{ | degree} = 3
$$

The above constraint enforces that the specified input and output rows must be present in the trace of the hash chiplet, and that they must be exactly $8 \cdot d - 1$ rows apart, where d is the depth of the node.

The effect of this operation on the rest of the stack is:
* **No change** starting from position 0.

MRUPDATE
The `MRUPDATE` operation computes a new root of a Merkle tree where a node at the specified position is updated to the specified value.

The stack is expected to be arranged as follows (from the top):
- old value of the node, 4 element (V in the below image)
- depth of the node, 1 element (d in the below image)
- index of the node, 1 element (i in the below image)
- current root of the tree, 4 elements (R in the below image)
- new value of the node, 4 element (NV in the below image)

The Merkle path for the node is expected to be provided by the prover non-deterministically (via merkle sets). At the end of the operation, the old node value is replaced with the new root value computed based on the provided path. Everything else on the stack remains the same. The diagram below illustrates this graphically.

![mrupdate](../../assets/design/stack/crypto_ops/MRUPDATE.png)

In the above, r (located in the helper register h_0) is the row address from the hash chiplet set by the prover non-deterministically.

For the `MRUPDATE` operation, we define input and output values as follows:

$$
v_{inputold} = \alpha_0 + \alpha_1 \cdot op_{mruold} + \alpha_2 \cdot h_0 + \alpha_3 \cdot s_5 + \sum_{j=0}^3\alpha_{j + 8} \cdot s_{3 - j}
$$

$$
v_{outputold} = \alpha_0 + \alpha_1 \cdot op_{rethash} + \alpha_2 \cdot (h_0 + 8 \cdot s_4 - 1) + \sum_{j=0}^3\alpha_{j + 8} \cdot s_{9 - j}
$$

$$
v_{inputnew} = \alpha_0 + \alpha_1 \cdot op_{mrunew} + \alpha_2 \cdot (h_0 + 8 \cdot s_4) + \alpha_3 \cdot s_5 + \sum_{j=0}^3\alpha_{j + 8} \cdot s_{13 - j}
$$

$$
v_{outputnew} = \alpha_0 + \alpha_1 \cdot op_{rethash} + \alpha_2 \cdot (h_0 + 2 \cdot 8 \cdot s_4 - 1) + \sum_{j=0}^3\alpha_{j + 8} \cdot s_{3 - j}'
$$

In the above, the first two expressions correspond to inputs and outputs for verifying the Merkle path between the old node value and the old tree root, while the last two expressions correspond to inputs and outputs for verifying the Merkle path between the new node value and the new tree root. The hash chiplet ensures the same set of sibling nodes are uses in both of these computations.

The op_{mruold}, op_{mrunew}, and $op_{rethash}$ are the unique [operation labels](../chiplets/main.md#operation-labels) used by the above computations.

$$
b_{chip}' \cdot v_{inputold} \cdot v_{outputold} \cdot v_{inputnew} \cdot v_{outputnew} = b_{chip} \text{ | degree} = 5
$$

The above constraint enforces that the specified input and output rows for both, the old and the new node/root combinations, must be present in the trace of the hash chiplet, and that they must be exactly $8 \cdot d - 1$ rows apart, where d is the depth of the node. It also ensures that the computation for the old node/root combination is immediately followed by the computation for the new node/root combination.

The effect of this operation on the rest of the stack is:
* **No change** for positions starting from 4.

FRIE2F4
The `FRIE2F4` operation performs FRI layer folding by a factor of 4 for FRI protocol executed in a degree 2 extension of the base field. It also performs several computations needed for checking correctness of the folding from the previous layer as well as simplifying folding of the next FRI layer.

The stack for the operation is expected to be arranged as follows:
- The first 8 stack elements contain 4 query points to be folded. Each point is represented by two field elements because points to be folded are in the extension field. We denote these points as $q_0 = (v_0, v_1)$, $q_1 = (v_2, v_3)$, $q_2 = (v_4, v_5)$, $q_3 = (v_6, v_7)$.
- The next element f_pos is the query position in the folded domain. It can be computed as $pos \mod n$, where pos is the position in the source domain, and n is size of the folded domain.
- The next element d_seg is a value indicating domain segment from which the position in the original domain was folded. It can be computed as $\lfloor \frac{pos}{n} \rfloor$. Since the size of the source domain is always 4 times bigger than the size of the folded domain, possible domain segment values can be 0, 1, 2, or 3.
- The next element poe is a power of initial domain generator which aid in a computation of the domain point x.
- The next two elements contain the result of the previous layer folding - a single element in the extension field denoted as $pe = (pe_0, pe_1)$.
- The next two elements specify a random verifier challenge α for the current layer defined as $\alpha = (a_0, a_1)$.
- The last element on the top of the stack ($cptr$) is expected to be a memory address of the layer currently being folded.

The diagram below illustrates stack transition for `FRIE2F4` operation.

![frie2f4](../../assets/design/stack/crypto_ops/FRIE2F4.png)

At the high-level, the operation does the following:
- Computes the domain value x based on values of poe and d_seg.
- Using x and α, folds the query values $q_0, ..., q_3$ into a single value r.
- Compares the previously folded value pe to the appropriate value of $q_0, ..., q_3$ to verify that the folding of the previous layer was done correctly.
- Computes the new value of poe as $poe' = poe^4$ (this is done in two steps to keep the constraint degree low).
- Increments the layer address pointer by 2.
- Shifts the stack by 1 to the left. This moves an element from the stack overflow table into the last position on the stack top.

To keep the degree of the constraints low, a number of intermediate values are used. Specifically, the operation relies on all 6 helper registers, and also uses the first 10 elements of the stack at the next state for degree reduction purposes. Thus, once the operation has been executed, the top 10 elements of the stack can be considered to be "garbage".

> TODO: add detailed constraint descriptions. See discussion [here](https://github.com/0xPolygonMiden/miden-vm/issues/567#issuecomment-1398088792).

The effect on the rest of the stack is:
* **Left shift** starting from position 16.
---
id: field_ops
title: Field Operations
sidebar_label: Field Operations
description: "AIR constraints for Miden VM field operations."
keywords:
  - docs
  - matic
  - polygon
  - wiki
  - miden
  - design
  - field
  - operation
image: https://wiki.polygon.technology/img/thumbnail/polygon-miden.png
---

In this section we describe the AIR constraints for Miden VM field operations (i.e., arithmetic operations over field elements).

ADD

Assume a and b are the elements at the top of the stack. The `ADD` operation computes $c \leftarrow (a + b)$. The diagram below illustrates this graphically.

![add](../../assets/design/stack/field_operations/ADD.png)

Stack transition for this operation must satisfy the following constraints:

$$
s_0' - (s_0 + s_1) = 0 \text{ | degree} = 1
$$

The effect on the rest of the stack is:
* **Left shift** starting from position 2.

NEG
Assume a is the element at the top of the stack. The `NEG` operation computes $b \leftarrow (-a)$. The diagram below illustrates this graphically.

![neg](../../assets/design/stack/field_operations/NEG.png)

Stack transition for this operation must satisfy the following constraints:

$$
s_0' + s_0 = 0 \text{ | degree} = 1
$$

The effect on the rest of the stack is:
* **No change** starting from position 1.

MUL
Assume a and b are the elements at the top of the stack. The `MUL` operation computes $c \leftarrow (a \cdot b)$. The diagram below illustrates this graphically.

![mul](../../assets/design/stack/field_operations/MUL.png)

Stack transition for this operation must satisfy the following constraints:

$$
s_0' - s_0 \cdot s_1 = 0 \text{ | degree} = 2
$$

The effect on the rest of the stack is:
* **Left shift** starting from position 2.

INV
Assume a is the element at the top of the stack. The `INV` operation computes $b \leftarrow (a^{-1})$. The diagram below illustrates this graphically.

![inv](../../assets/design/stack/field_operations/INV.png)

Stack transition for this operation must satisfy the following constraints:

$$
1 - s_0' \cdot s_0 = 0 \text{ | degree} = 2
$$

Note that the above constraint can be satisfied only if the value in $s_0 \neq 0$.

The effect on the rest of the stack is:
* **No change** starting from position 1.

INCR
Assume a is the element at the top of the stack. The `INCR` operation computes $b \leftarrow (a+1)$. The diagram below illustrates this graphically.

![incr](../../assets/design/stack/field_operations/INCR.png)

Stack transition for this operation must satisfy the following constraints:

$$
s_0' - (s_0 + 1) = 0 \text{ | degree} = 1
$$

The effect on the rest of the stack is:
* **No change** starting from position 1.

NOT
Assume a is a binary value at the top of the stack. The `NOT` operation computes $b \leftarrow (\lnot a)$. The diagram below

illustrates this graphically.

![not](../../assets/design/stack/field_operations/NOT.png)

Stack transition for this operation must satisfy the following constraints:

$$
s_0^2 - s_0 = 0 \text{ | degree} = 2
$$

$$
s_0' - (1 - s_0) = 0 \text{ | degree} = 1
$$

The first constraint ensures that the value in s_0 is binary, and the second constraint ensures the correctness of the boolean `NOT` operation.

The effect on the rest of the stack is:
* **No change** starting from position 1.

AND
Assume a and b are binary values at the top of the stack. The `AND` operation computes $c \leftarrow (a \land b)$. The diagram below illustrates this graphically.

![and](../../assets/design/stack/field_operations/AND.png)

Stack transition for this operation must satisfy the following constraints:

$$
s_i^2 - s_i = 0 \text{ for } i \in \{0, 1\} \text{ | degree} = 2
$$

$$
s_0' - s_0 \cdot s_1 = 0 \text{ | degree} = 2
$$

The first two constraints ensure that the value in s_0 and s_1 are binary, and the third constraint ensures the correctness of the boolean `AND` operation.

The effect on the rest of the stack is:
* **Left shift** starting from position 2.

OR
Assume a and b are binary values at the top of the stack. The `OR` operation computes $c \leftarrow (a \lor b)$ The diagram below illustrates this graphically.

![or](../../assets/design/stack/field_operations/OR.png)

Stack transition for this operation must satisfy the following constraints:

$$
s_i^2 - s_i = 0 \text{ for } i \in \{0, 1\} \text{ | degree} = 2
$$

$$
s_{0}' - (s_{1} + s_{0} - s_{1} \cdot s_{0}) = 0 \text{ | degree} = 2
$$

The first two constraints ensure that the value in s_0 and s_1 are binary, and the third constraint ensures the correctness of the boolean `OR` operation.

The effect on the rest of the stack is:
* **Left shift** starting from position 2.

EQ
Assume a and b are the elements at the top of the stack. The `EQ` operation computes c such that $c = 1$ if $a = b$, and 0 otherwise. The diagram below illustrates this graphically.

![eq](../../assets/design/stack/field_operations/EQ.png)

Stack transition for this operation must satisfy the following constraints:

$$
s_0' \cdot (s_0 - s_1) = 0 \text{ | degree} = 2
$$

$$
s_0' - (1 - (s_0 - s_1) \cdot h_0) = 0 \text{ | degree} = 2

$$

To satisfy the above constraints, the prover must populate the value of helper register h_0 as follows:
* If $s_0 \neq s_1$, set $h_0 = \frac{1}{s_0 - s_1}$.
* Otherwise, set h_0 to any value (e.g., 0).

The effect on the rest of the stack is:
* **Left shift** starting from position 2.

EQZ
Assume a is the element at the top of the stack. The `EQZ` operation computes b such that $b = 1$ if $a = 0$, and 0 otherwise. The diagram below illustrates this graphically.

![eqz](../../assets/design/stack/field_operations/EQZ.png)

Stack transition for this operation must satisfy the following constraints:

$$
s_0' \cdot s_0 = 0 \text{ | degree} = 2
$$

$$
s_0' - (1 - s_0 \cdot h_0) = 0 \text{ | degree} = 2
$$

To satisfy the above constraints, the prover must populate the value of helper register h_0 as follows:
* If $s_0 \neq 0$, set $h_0 = \frac{1}{s_0}$.
* Otherwise, set h_0 to any value (e.g., 0).

The effect on the rest of the stack is:
* **No change** starting from position 1.

EXPACC
The `EXPACC` operation pops top 4 elements from the top of the stack, performs a single round of exponent aggregation, and pushes the resulting 4 values onto the stack. The diagram below illustrates this graphically.

![expacc](../../assets/design/stack/field_operations/EXPACC.png)

Stack transition for this operation must satisfy the following constraints:

`bit` should be a binary.

$$
s_0'^{2} - s_0' = 0 \text{ | degree} = 2
$$

The `exp` in the next frame should be the square of the `exp` in the current frame.

$$
s_1' - s_1^{2} = 0 \text{ | degree} = 2
$$

The value `val` in the helper register is computed correctly using the `bit` and `exp` in next and current frame respectively.

$$
h_0 - ((s_1 - 1) * s_0' + 1) = 0 \text{ | degree} = 2
$$

The `acc` in the next frame is the product of `val` and `acc` in the current frame.

$$
s_2' - s_2 * h_0 = 0 \text{ | degree} = 2
$$

`b` in the next frame is the right shift of `b` in the current frame.

$$
s_3' - (s_3 * 2 + s_0') = 0 \text{ | degree} = 1
$$

The effect on the rest of the stack is:
* **No change** starting from position 4.

EXT2MUL
The `EXT2MUL` operation pops top 4 values from the top of the stack, performs multiplication between the two extension field elements, and pushes the resulting 4 values onto the stack. The diagram below illustrates this graphically.

![ext2mul](../../assets/design/stack/field_operations/EXT2MUL.png)

Stack transition for this operation must satisfy the following constraints:

The first stack element should be unchanged in the next frame.

$$
s_0' - s_0 = 0 \text{ | degree} = 1
$$

The second stack element should be unchanged in the next frame.

$$
s_1' - s_1 = 0 \text{ | degree} = 1
$$

The third stack element should satisfy the following constraint.

$$
s_2' - (s_0 + s_1) \cdot (s_2 + s_3) + s_0 \cdot s_2 = 0 \text{ | degree} = 2
$$

The fourth stack element should satisfy the following constraint.

$$
s_3' - s_1 \cdot s_3 + 2 \cdot s_0 \cdot s_2 = 0 \text{ | degree} = 2
$$

The effect on the rest of the stack is:
* **No change** starting from position 4.
---
id: io_ops
title: Input/Output Operations
sidebar_label: I/O Operations
description: "AIR constraints for Miden VM input / output operations."
keywords:
  - docs
  - matic
  - polygon
  - wiki
  - miden
  - design
  - input
  - output
image: https://wiki.polygon.technology/img/thumbnail/polygon-miden.png
---

In this section we describe the AIR constraints for Miden VM input / output operations. These operations move values between the stack and other components of the VM such as program code (i.e., decoder), memory, and advice provider.

PUSH
The `PUSH` operation pushes the provided immediate value onto the stack (i.e., sets the value of s_0 register). Currently, it is the only operation in Miden VM which carries an immediate value. The semantics of this operation are explained in the [decoder section](../decoder/main.html#handling-immediate-values).

The effect of this operation on the rest of the stack is:
* **Right shift** starting from position 0.

SDEPTH
Assume a is the current depth of the stack stored in the stack bookkeeping register b_0 (as described [here](./main.md#stack-representation)). The `SDEPTH` pushes a onto the stack. The diagram below illustrates this graphically.

![sdepth](../../assets/design/stack/io_ops/SDEPTH.png)

Stack transition for this operation must satisfy the following constraints:

$$
s_0' - b_0 = 0 \text{ | degree} = 1
$$

The effect of this operation on the rest of the stack is:
* **Right shift** starting from position 0.

ADVPOP
Assume a is an element at the top of the advice stack. The `ADVPOP` operation removes a from the advice stack and pushes it onto the operand stack. The diagram below illustrates this graphically.

![advpop](../../assets/design/stack/io_ops/ADVPOP.png)

The `ADVPOP` operation does not impose any constraints against the first element of the operand stack.

The effect of this operation on the rest of the operand stack is:
* **Right shift** starting from position 0.

ADVPOPW
Assume a, b, c, and d, are the elements at the top of the advice stack (with a being on top). The `ADVPOPW` operation removes these elements from the advice stack and puts them onto the operand stack by overwriting the top 4 stack elements. The diagram below illustrates this graphically.

![advpopw](../../assets/design/stack/io_ops/ADVPOPW.png)

The `ADVPOPW` operation does not impose any constraints against the top 4 elements of the operand stack.

The effect of this operation on the rest of the operand stack is:
* **No change** starting from position 4.

Memory access operations
Miden VM exposes several operations for reading from and writing to random access memory. Memory in Miden VM is managed by the [Memory chiplet](../chiplets/memory.md).

Communication between the stack and the memory chiplet is accomplished via the chiplet bus b_{chip}. To make requests to the chiplet bus we need to divide its current value by the value representing memory access request. The structure of memory access request value is described [here](../chiplets/memory.md#memory-row-value).

To enforce the correctness of memory access, we can use the following constraint:

$$
b_{chip}' \cdot u_{mem} = b_{chip} \text{ | degree} = 2
$$

In the above, u_{mem} is the value of memory access request. Thus, to describe AIR constraint for memory operations, it is sufficient to describe how u_{mem} is computed. We do this in the following sections.

MLOADW
Assume that the word with elements v_0, v_1, v_2, v_3 is located in memory at address a. The `MLOADW` operation pops an element off the stack, interprets it as a memory address, and replaces the remaining 4 elements at the top of the stack with values located at the specified address. The diagram below illustrates this graphically.

![mloadw](../../assets/design/stack/io_ops/MLOADW.png)

To simplify description of the memory access request value, we first define a variable for the value that represents the state of memory after the operation:

$$
v = \sum_{i=0}^3\alpha_{i+5} \cdot s_{3-i}'
$$

Using the above variable, we define the value representing the memory access request as follows:

$$
u_{mem} = \alpha_0 + \alpha_1 \cdot op_{mem_read} + \alpha_3 \cdot s_0 + \alpha_4 \cdot clk + v
$$

In the above:
- op_{mem_read} is the unique [operation label](../chiplets/main.md#operation-labels) of the memory read operation.
- s_0 is the memory address from which the values are to be loaded onto the stack.
- clk is the current clock cycle of the VM.

Note that α_2 term is skipped because currently memory context value is always 0.

The effect of this operation on the rest of the stack is:
* **Left shift** starting from position 5.

MLOAD
Assume that the word with elements v_0, v_1, v_2, v_3 is located in memory at address a. The `MLOAD` operation pops an element off the stack, interprets it as a memory address, and pushes the first element of the word located at the specified address to the stack. The diagram below illustrates this graphically.

![mload](../../assets/design/stack/io_ops/MLOAD.png)

To simplify description of the memory access request value, we first define a variable for the value that represents the state of memory after the operation:

$$
v = \alpha_5 \cdot s_0' + \sum_{i=1}^3\alpha_{i+5} \cdot h_{3-i}'
$$

Note: the values in registers h_0, h_1, h_2 are set by the prover non-deterministically.

Using the above variable, we define the value representing the memory access request as follows:

$$
u_{mem} = \alpha_0 + \alpha_1 \cdot op_{mem_read} + \alpha_3 \cdot s_0 + \alpha_4 \cdot clk + v
$$

In the above:
- op_{mem_read} is the unique [operation label](../chiplets/main.md#operation-labels) of the memory read operation.
- s_0 is the memory address from which the value is to be loaded onto the stack.
- clk is the current clock cycle of the VM.

Note that α_2 term is skipped because currently memory context value is always 0.

The effect of this operation on the rest of the stack is:
* **No change** starting from position 1.

MSTOREW
The `MSTOREW` operation pops an element off the stack, interprets it as a memory address, and writes the remaining 4 elements at the top of the stack into memory at the specified address. The stored elements are not removed from the stack. The diagram below illustrates this graphically.

![mstorew](../../assets/design/stack/io_ops/MSTOREW.png)

After the operation the contents of memory at address a would be set to v_0, v_1, v_2, v_3.

To simplify description of the memory access request value, we first define a variable for the value that represents the state of memory after the operation:

$$
v = \sum_{i=0}^3\alpha_{i+5} \cdot s_{3-i}'
$$

Using the above variable, we define the value representing the memory access request as follows:

$$
u_{mem} = \alpha_0 + \alpha_1 \cdot op_{mem_write} + \alpha_3 \cdot s_0 + \alpha_4 \cdot clk + v
$$

In the above:
- op_{mem_write} is the unique [operation label](../chiplets/main.md#operation-labels) of the memory write operation.
- s_0 is the memory address into which the values from the stack are to be saved.
- clk is the current clock cycle of the VM.

Note that α_2 term is skipped because currently memory context value is always 0.

The effect of this operation on the rest of the stack is:
* **Left shift** starting from position 1.

MSTORE
The `MSTORE` operation pops an element off the stack, interprets it as a memory address, and writes the remaining element at the top of the stack into the first element of the word located at the specified memory address. The remaining 3 elements of the word are not affected. The diagram below illustrates this graphically.

![mstore](../../assets/design/stack/io_ops/MSTORE.png)

After the operation the contents of memory at address a would be set to b, v_1, v_2, v_3.

To simplify description of the memory access request value, we first define a variable for the value that represents the state of memory after the operation:

$$
v = \alpha_5 \cdot s_0' + \sum_{i=1}^3\alpha_{i+5} \cdot h_{3-i}'
$$

Note: the values in registers h_0, h_1, h_2 are set by the prover non-deterministically.

Using the above variable, we define the value representing the memory access request as follows:

$$
u_{mem} = \alpha_0 + \alpha_1 \cdot op_{mem_write} + \alpha_3 \cdot s_0 + \alpha_4 \cdot clk + v
$$

In the above:
- $op_{mem_write} $ is the unique [operation label](../chiplets/main.md#operation-labels) of the memory write operation.
- s_0 is the memory address into which the value from the stack is to be saved.
- clk is the current clock cycle of the VM.

Note that α_2 term is skipped because currently memory context value is always 0.

The effect of this operation on the rest of the stack is:
* **Left shift** starting from position 1.
---
id: main
title: Operand Stack
sidebar_label: Operand Stack
description: "Stack rules and restrictions."
keywords:
  - docs
  - matic
  - polygon
  - wiki
  - miden
  - design
  - operand
  - stack
image: https://wiki.polygon.technology/img/thumbnail/polygon-miden.png
---

Miden VM is a stack machine. The stack is a push-down stack of practically unlimited depth (in practical terms, the depth will never exceed 2^{32}), but only the top 16 items are directly accessible to the VM. Items on the stack are elements in a prime field with modulus $2^{64} - 2^{32} + 1$.

To keep the constraint system for the stack manageable, we impose the following rules:
1. All operations executed on the VM can shift the stack by at most one item. That is, the end result of an operation must be that the stack shrinks by one item, grows by one item, or the number of items on the stack stays the same.
2. Stack depth must always be greater than or equal to 16. At the start of program execution, the stack is initialized with exactly 16 input values, all of which could be 0's.
3. By the end of program execution, exactly 16 items must remain on the stack (again, all of them could be 0's). These items comprise the output of the program.

To ensure that managing stack depth does not impose significant burden, we adopt the following rule:

* When the stack depth is 16, removing additional items from the stack does not change its depth. To keep the depth at 16, 0's are inserted into the deep end of the stack for each removed item.

Stack representation
The VM allocates 19 trace columns for the stack. The layout of the columns is illustrated below.

The meaning of the above columns is as follows:
* $s_0 ... s_{15}$ are the columns representing the top 16 slots of the stack.
* Column b_0 contains the number of items on the stack (i.e., the stack depth). In the above picture, there are 16 items on the stacks, so $b_0 = 16$.
* Column b_1 contains an address of a row in the "overflow table" in which we'll store the data that doesn't fit into the top 16 slots. When $b_1 = 0$, it means that all stack data fits into the top 16 slots of the stack.
* Helper column h_0 is used to ensure that stack depth does not drop below 16. Values in this column are set by the prover non-deterministically to $\frac{1}{b_0 - 16}$ when $b_0 \neq 16$, and to any other value otherwise.

Overflow table
To keep track of the data which doesn't fit into the top 16 stack slots, we'll use an overflow table. This will be a [virtual table](../multiset.md#virtual-tables). To represent this table, we'll use a single auxiliary column p_1.

The table itself can be thought of as having 3 columns as illustrated below.

The meaning of the columns is as follows:
* Column t_0 contains row address. Every address in the table must be unique.
* Column t_1 contains the value that overflowed the stack.
* Column t_2 contains the address of the row containing the value that overflowed the stack right before the value in the current row. For example, in the picture above, first value a overflowed the stack, then b overflowed the stack, and then value c overflowed the stack. Thus, row with value b points back to the row with value a, and row with value c points back to the row with value b.

To reduce a table row to a single value, we'll compute a randomized product of column values as follows:

$$
r_i = \alpha_0 + \alpha_1 \cdot t_{0, i} + \alpha_2 \cdot t_{1, i} + \alpha_3 \cdot t_{2, i}
$$

Then, when row i is added to the table, we'll update the value in the p_1 column like so:

$$

```
p_1' = p_1 \cdot r_i
$$

Analogously, when row $i$ is removed from the table, we'll update the value in column $p_1$ like so:

$$
p_1' = \frac{p_1}{r_i}
$$

The initial value of $p_1$ is set to $1$. Thus, if by the time Miden VM finishes executing a program the table is empty (we added and then removed exactly the same set of rows), $p_1$ will also be equal to $1$.

There are a couple of other rules we'll need to enforce:
* We can delete a row only after the row has been inserted into the table.
* We can't insert a row with the same address twice into the table (even if the row was inserted and then deleted).

How these are enforced will be described a bit later.

## Right shift
If an operation adds data to the stack, we say that the operation caused a right shift. For example, `PUSH` and `DUP` operations cause a right shift. Graphically, this looks like so:

![](../../assets/design/stack/stack_right_shift.png)

Here, we pushed value $v_{17}$ onto the stack. All other values on the stack are shifted by one slot to the right and the stack depth increases by $1$. There is not enough space at the top of the stack for all $17$ values, thus, $v_1$ needs to be moved to the overflow table.

To do this, we need to rely on another column: $k_0$. This is a system column which keeps track of the current VM cycle. The value in this column is simply incremented by $1$ with every step.

The row we want to add to the overflow table is defined by tuple $(clk, v1, 0)$, and after it is added, the table would look like so:

![](../../assets/design/stack/stack_overflow_table_post_1_right_shift.png)

The reason we use VM clock cycle as row address is that the clock cycle is guaranteed to be unique, and thus, the same row can not be added to the table twice.

Let's push another item onto the stack:

![](../../assets/design/stack/stack_overflow_push_2nd_item.png)

Again, as we push $v_{18}$ onto the stack, all items on the stack are shifted to the right, and now $v_2$ needs to be moved to the overflow table. The tuple we want to insert into the table now is $(clk+1, v2, clk)$. After the operation, the overflow table will look like so:

![](../../assets/design/stack/stack_overflow_table_post_2_right_shift.png)

Notice that $t_2$ for row which contains value $v_2$ points back to the row with address $clk$.

Overall, during a right shift we do the following:
* Increment stack depth by $1$.
* Shift stack columns $s_0, ..., s_{14}$ right by $1$ slot.
* Add a row to the overflow table described by tuple $(k_0, s_{15}, b_0)$.
* Set the next value of $b_1$ to the current value of $k_0$.

Also, as mentioned previously, the prover sets values in $h_0$ non-deterministically to $\frac{1}{b_0 - 16}$.

## Left shift
If an operation removes an item from the stack, we say that the operation caused a left shift. For example, a `DROP` operation causes a left shift. Assuming the stack is in the state we left it at the end of the previous section, graphically, this looks like so:

![](../../assets/design/stack/stack_1st_left_shift.png)

Overall, the during the left shift we do the following:

* When stack depth is greater than $16$:
  * Decrement stack depth by $1$.
  * Shift stack columns $s_1, ..., s_{15}$ left by $1$ slot.
  * Remove a row from the overflow table with $t_0$ equal to the current value of $b_1$.
  * Set the next value of $s_{15}$ to the value in $t_1$ of the removed overflow table row.
  * Set the next value of $b_1$ to the value in $t_2$ of the removed overflow table row.
* When the stack depth is equal to $16$:
  * Keep the stack depth the same.
  * Shift stack columns $s_1, ..., s_{15}$ left by $1$ slot.
  * Set the value of $s_{15}$ to $0$.
```

```
  * Set the value to $h_0$ to $0$ (or any other value).

If the stack depth becomes (or remains) $16$, the prover can set $h_0$ to any value (e.g., $0$). But if the depth is greater than
$16$ the prover sets $h_0$ to $\frac{1}{b_0 - 16}$.

## AIR Constraints
To simplify constraint descriptions, we'll assume that the VM exposes two binary flag values described below.

| Flag      | Degree | Description |
| --------- | ------ | ----------- |
| $f_{shr}$ | 6      | When this flag is set to $1$, the instruction executing on the VM is performing a "right shift". |
| $f_{shl}$ | 5      | When this flag is set to $1$, the instruction executing on the VM is performing a "left shift". |

These flags are mutually exclusive. That is, if $f_{shl}=1$, then $f_{shr}=0$ and vice versa. However, both flags can be set to
$0$ simultaneously. This happens when the executed instruction does not shift the stack. How these flags are computed is described
[here](./op_constraints.md).

### Stack overflow flag
Additionally, we'll define a flag to indicate whether the overflow table contains values. This flag will be set to $0$ when the
overflow table is empty, and to $1$ otherwise (i.e., when stack depth $>16$). This flag can be computed as follows:

$$
f_{ov} = (b_0 - 16) \cdot h_0 \text{ | degree} = 2
$$

To ensure that this flag is set correctly, we need to impose the following constraint:

$$
(1 - f_{ov}) \cdot (b_0 - 16) = 0 \text{ | degree} = 3
$$

The above constraint can be satisfied only when either of the following holds:

* $b_0 = 16$, in which case $f_{ov}$ evaluates to $0$, regardless of the value of $h_0$.
* $f_{ov} = 1$, in which case $b_0$ cannot be equal to $16$ (and $h_0$ must be set to $\frac{1}{b_0 - 16}$).

### Stack depth constraints
To make sure stack depth column $b_0$ is updated correctly, we need to impose the following constraints:

| Condition   | Constraint__     | Description |
| ----------- | ---------------- | ----------- |
| $f_{shr}=1$ | $b'_0 = b_0 + 1$ | When the stack is shifted to the right, stack depth should be incremented by $1$. |
| $f_{shl}=1$ <br></br> $f_{ov}=1$ | $b'_0 = b_0 - 1$ | When the stack is shifted to the left and the overflow table is not empty,
stack depth should be decremented by $1$. |
| otherwise   | $b'_0 = b_0$     | In all other cases, stack depth should not change. |

We can combine the above constraints into a single expression as follows:

$$
b'_0 - b_0 + f_{shl} \cdot f_{ov} - f_{shr} = 0 \text{ | degree} = 7
$$

### Overflow table constraints

When the stack is shifted to the right, a tuple $(k_0, s_{15}, b_1)$ should be added to the overflow table. We will denote value
of the row to be added to the table as follows:

$$
v = \alpha_0 + \alpha_1 \cdot k_0 + \alpha_2 \cdot s_{15} + \alpha_3 \cdot b_1
$$

When the stack is shifted to the left, a tuple $(b_1, s'_{15}, b'_1)$ should be removed from the overflow table. We will denote
value of the row to be removed from the table as follows.

$$
u = \alpha_0 + \alpha_1 \cdot b_1 + \alpha_2 \cdot s'_{15} + \alpha_3 \cdot b'_1
$$

Using the above variables, we can ensure that right and left shifts update the overflow table correctly by enforcing the following
constraint:

$$
p_1' \cdot (u \cdot f_{shl} \cdot f_{ov} + 1 - f_{shl} \cdot f_{ov}) = p_1 \cdot (v \cdot f_{shr} + 1 - f_{shr}) \text{ | degree}
= 9
$$

The above constraint reduces to the following under various flag conditions:
```

Condition	Applied constraint
$f_{shl}=1$, $f_{shr}=0$, $f_{ov}=0$	$p_1' = p_1$
$f_{shl}=1$, $f_{shr}=0$, $f_{ov}=1$	$p_1' \cdot u = p_1$
$f_{shl}=0$, $f_{shr}=1$, $f_{ov}=1 \text{ or } 0$	$p_1' = p_1 \cdot v$
$f_{shl}=0$, $f_{shr}=0$, $f_{ov}=1 \text{ or } 0$	$p_1' = p_1$

Notice that in the case of the left shift, the constraint forces the prover to set the next values of s_{15} and b_1 to values t_1 and t_2 of the row removed from the overflow table.

In case of a right shift, we also need to make sure that the next value of b_1 is set to the current value of k_0. This can be done with the following constraint:

$$
f_{shr} \cdot (b'_1 - k_0) = 0 \text{ | degree} = 7
$$

In case of a left shift, when the overflow table is empty, we need to make sure that a 0 is "shifted in" from the right (i.e., s_{15} is set to 0). This can be done with the following constraint:

$$
f_{shl} \cdot (1 - f_{ov}) \cdot s_{15}' = 0 \text{ | degree} = 8
$$

Boundary constraints
In addition to the constraints described above, we also need to enforce the following boundary constraints:
* $b_0 = 16$ at the first and at the last row of execution trace.
* $b_1 = 0$ at the first and at the last row of execution trace.
* $p_1 = 1$ at the first and at the last row of execution trace.
---
id: op_constraints
title: Stack Operation Constraints
sidebar_label: Stack Operation Constraints
description: Operation-specific constraints.
keywords:
  - docs
  - matic
  - polygon
  - wiki
  - miden
  - design
  - operation
image: https://wiki.polygon.technology/img/thumbnail/polygon-miden.png
---

In addition to the constraints described in the previous section, we need to impose constraints to check that each VM operation is executed correctly.

For this purpose the VM exposes a set of operation-specific flags. These flags are set to 1 when a given operation is executed, and to 0 otherwise. The naming convention for these flags is f_{opname}. For example, f_{dup} would be set to 1 when `DUP` operation is executed, and to 0 otherwise. Operation flags are discussed in detail in the section [below](#operation-flags).

To describe how operation-specific constraints work, let's use an example with `DUP` operation. This operation pushes a copy of the top stack item onto the stack. The constraints we need to impose for this operation are as follows:

$$
f_{dup} \cdot (s'_0 - s_0) = 0 \\
f_{dup} \cdot (s'_{i+1} - s_i) = 0 \ \text{ for } i \in \{0, .., 14\}
$$

The first constraint enforces that the top stack item in the next row is the same as the top stack item in the current row. The second constraint enforces that all stack items (starting from item 0) are shifted to the right by 1. We also need to impose all the constraints discussed in the previous section, be we omit them here.

Let's write similar constraints for `DUP1` operation, which pushes a copy of the second stack item onto the stack:

$$
f_{dup1} \cdot (s'_0 - s_1) = 0 \\
f_{dup1} \cdot (s'_{i+1} - s_i) = 0 \ \text{ for } i \in \{0, .., 14\}
$$

It is easy to notice that while the first constraint changed, the second constraint remained the same - i.e., we are still just shifting the stack to the right.

In fact, for most operations it makes sense to make a distinction between constraints unique to the operation vs. more general constraints which enforce correct behavior for the stack items not affected by the operation. In the subsequent sections we describe in detail only the former constraints, and provide high-level descriptions of the more general constraints. Specifically, we indicate how the operation affects the rest of the stack (e.g., shifts right starting from position 0).

Operation flags
As mentioned above, operation flags are used as selectors to enforce operation-specific constraints. That is, they turn on relevant constraints for a given operation. In total, the VM provides 88 unique operations, and thus, there are 88 operation flags (not all of them currently used).

Operation flags are mutually exclusive. That is, if one flag is set to 1, all other flags are set to 0. Also, one of the flags is always guaranteed to be set to 1.

To compute values of operation flags we use _op bits_ registers located in the [decoder](../decoder/main.md#decoder-trace). These registers contain binary representations of operation codes (opcodes). Each opcode consists of 7 bits, and thus, there are 7 _op bits_ registers. We denote these registers as $b_0, ..., b_6$. The values are computed by multiplying the op bit registers in various combinations.

For example, the value of the flag for `NOOP`, which is encoded as `0000000`, is computed as follows:

$$
f_{noop} = (1 - b_0) \cdot (1 - b_1) \cdot (1 - b_2) \cdot (1 - b_3) \cdot (1 - b_4) \cdot (1 - b_5) \cdot (1 - b_6)
$$

While the value of the `DROP` operation, which is encoded as `0101001` is computed as follows:

$$
f_{drop} = (1 - b_0) \cdot b_1 \cdot (1 - b_2) \cdot b_3 \cdot (1 - b_4) \cdot (1 - b_5) \cdot b_6
$$

As can be seen from above, the degree for both of these flags is 7. Since degree of constraints in Miden VM can go up to 9, this means that operation-specific constraints cannot exceed degree 2. However, there are some operations which require constraints of higher degree (e.g., 3 or even 5). To support such constraints, we adopt the following scheme.

We organize the operations into 3 groups as shown below and also introduce an extra register for degree reduction:

b_6	b_5	b_4	b_3	b_2	b_1	b_0	extra	# of ops	degree
0	x	x	x	x	x	x	0	64	7
1	0	x	x	x	x	-	0	16	6
1	1	x	x	x	-	-	1	8	4

In the above:
* Operation flags for operations in the first group (with prefix `0`), are computed using all 7 op bits, and thus their degree is 7.
* Operation flags for operations in the second group (with prefix `10`), are computed using only the first 6 op bits, and thus, their degree is 6.
* Operation flags for operations in the third group (with prefix `11`), are computed using only the first 5 op bits, and we also use the extra register (which is set to $b_6 \cdot b_5$) to reduce the degree by 1. Thus, the degree of op flags in this group is 4.

How operations are distributed between these 3 groups is described in the sections below.

No stack shift operations
This group contains 32 operations which do not shift the stack (this is almost all such operations). Since the op flag degree for these operations is 7, constraints for these operations cannot exceed degree 2.

Operation	Opcode value	Binary encoding	Operation group	Flag degree
`NOOP`	0	`000_0000`	[System ops](./system_ops.md)	7
`EQZ `	1	`000_0001`	[Field ops](./field_ops.md)	7
`NEG`	2	`000_0010`	[Field ops](./field_ops.md)	7
`INV`	3	`000_0011`	[Field ops](./field_ops.md)	7
`INCR`	4	`000_0100`	[Field ops](./field_ops.md)	7
`NOT`	5	`000_0101`	[Field ops](./field_ops.md)	7
`FMPADD`	6	`000_0110`	[System ops](./system_ops.md)	7
`MLOAD`	7	`000_0111`	[I/O ops](./io_ops.md)	7
`SWAP`	8	`000_1000`	[Stack ops](./stack_ops.md)	7
`CALLER`	9	`000_1001`	[System ops](./system_ops.md)	7
`MOVUP2`	10	`000_1010`	[Stack ops](./stack_ops.md)	7
`MOVDN2`	11	`000_1011`	[Stack ops](./stack_ops.md)	7
`MOVUP3`	12	`000_1100`	[Stack ops](./stack_ops.md)	7
`MOVDN3`	13	`000_1101`	[Stack ops](./stack_ops.md)	7
`ADVPOPW`	14	`000_1110`	[I/O ops](./io_ops.md)	7
`EXPACC`	15	`000_1111`	[Field ops](./field_ops.md)	7
`MOVUP4`	16	`001_0000`	[Stack ops](./stack_ops.md)	7
`MOVDN4`	17	`001_0001`	[Stack ops](./stack_ops.md)	7
`MOVUP5`	18	`001_0010`	[Stack ops](./stack_ops.md)	7
`MOVDN5`	19	`001_0011`	[Stack ops](./stack_ops.md)	7
`MOVUP6`	20	`001_0100`	[Stack ops](./stack_ops.md)	7
`MOVDN6`	21	`001_0101`	[Stack ops](./stack_ops.md)	7
`MOVUP7`	22	`001_0110`	[Stack ops](./stack_ops.md)	7

`MOVDN7`	23	`001_0111`	[Stack ops](./stack_ops.md)	7
`SWAPW`	24	`001_1000`	[Stack ops](./stack_ops.md)	7
`EXT2MUL`	25	`001_1001`	[Field ops](./field_ops.md)	7
`MOVUP8`	26	`001_1010`	[Stack ops](./stack_ops.md)	7
`MOVDN8`	27	`001_1011`	[Stack ops](./stack_ops.md)	7
`SWAPW2`	28	`001_1100`	[Stack ops](./stack_ops.md)	7
`SWAPW3`	29	`001_1101`	[Stack ops](./stack_ops.md)	7
`SWAPDW`	30	`001_1110`	[Stack ops](./stack_ops.md)	7
`<unused>`	31	`001_1111`		7

Left stack shift operations
This group contains 16 operations which shift the stack to the left (i.e., remove an item from the stack). Most of left-shift operations are contained in this group. Since the op flag degree for these operations is 7, constraints for these operations cannot exceed degree 2.

Operation	Opcode value	Binary encoding	Operation group	Flag degree
`ASSERT`	32	`010_0000`	[System ops](./system_ops.md)	7
`EQ`	33	`010_0001`	[Field ops](./field_ops.md)	7
`ADD`	34	`010_0010`	[Field ops](./field_ops.md)	7
`MUL`	35	`010_0011`	[Field ops](./field_ops.md)	7
`AND`	36	`010_0100`	[Field ops](./field_ops.md)	7
`OR`	37	`010_0101`	[Field ops](./field_ops.md)	7
`U32AND`	38	`010_0110`	[u32 ops](./u32_ops.md)	7
`U32XOR`	39	`010_0111`	[u32 ops](./u32_ops.md)	7
`FRIE2F4`	40	`010_1000`	[Crypto ops](./crypto_ops.md)	7
`DROP`	41	`010_1001`	[Stack ops](./stack_ops.md)	7
`CSWAP`	42	`010_1010`	[Stack ops](./stack_ops.md)	7
`CSWAPW`	43	`010_1011`	[Stack ops](./stack_ops.md)	7
`MLOADW`	44	`010_1100`	[I/O ops](./io_ops.md)	7
`MSTORE`	45	`010_1101`	[I/O ops](./io_ops.md)	7
`MSTOREW`	46	`010_1110`	[I/O ops](./io_ops.md)	7
`FMPUPDATE`	47	`010_1111`	[System ops](./system_ops.md)	7

Right stack shift operations
This group contains 16 operations which shift the stack to the right (i.e., push a new item onto the stack). Most of right-shift operations are contained in this group. Since the op flag degree for these operations is 7, constraints for these operations cannot exceed degree 2.

Operation	Opcode value	Binary encoding	Operation group	Flag degree
`PAD`	48	`011_0000`	[Stack ops](./stack_ops.md)	7
`DUP`	49	`011_0001`	[Stack ops](./stack_ops.md)	7
`DUP1`	50	`011_0010`	[Stack ops](./stack_ops.md)	7
`DUP2`	51	`011_0011`	[Stack ops](./stack_ops.md)	7
`DUP3`	52	`011_0100`	[Stack ops](./stack_ops.md)	7
`DUP4`	53	`011_0101`	[Stack ops](./stack_ops.md)	7
`DUP5`	54	`011_0110`	[Stack ops](./stack_ops.md)	7
`DUP6`	55	`011_0111`	[Stack ops](./stack_ops.md)	7
`DUP7`	56	`011_1000`	[Stack ops](./stack_ops.md)	7
`DUP9`	57	`011_1001`	[Stack ops](./stack_ops.md)	7
`DUP11`	58	`011_1010`	[Stack ops](./stack_ops.md)	7
`DUP13`	59	`011_1011`	[Stack ops](./stack_ops.md)	7
`DUP15`	60	`011_1100`	[Stack ops](./stack_ops.md)	7
`ADVPOP`	61	`011_1101`	[Stack ops](./io_ops.md)	7
`SDEPTH`	62	`011_1110`	[I/O ops](./io_ops.md)	7
`CLK`	63	`011_1111`	[System ops](./system_ops.md)	7

u32 operations
This group contains 8 u32 operations. These operations are grouped together because all of them require range checks. The constraints for range checks are of degree 5, however, since all these operations require them, we can define a flag with common prefix `100` to serve as a selector for the range check constraints. The value of this flag is computed as follows:

$$
f_{u32rc} = b_6 \cdot (1 - b_5) \cdot (1 - b_4)
$$

The degree of this flag is 3, which is acceptable for a selector for degree 5 constraints.

Operation	Opcode value	Binary encoding	Operation group	Flag degree
`U32ADD`	64	`100_0000`	[u32 ops](./u32_ops.md)	6
`U32SUB`	66	`100_0010`	[u32 ops](./u32_ops.md)	6
`U32MUL`	68	`100_0100`	[u32 ops](./u32_ops.md)	6
`U32DIV`	70	`100_0110`	[u32 ops](./u32_ops.md)	6
`U32SPLIT`	72	`100_1000`	[u32 ops](./u32_ops.md)	6
`U32ASSERT2`	74	`100_1010`	[u32 ops](./u32_ops.md)	6
`U32ADD3`	76	`100_1100`	[u32 ops](./u32_ops.md)	6

| `U32MADD` | 78 | `100_1110` | [u32 ops](./u32_ops.md) | 6 |

As mentioned previously, the last bit of the opcode is not used in computation of the flag for these operations. We force this bit to always be set to 0 with the following constraint:

$$
b_6 \cdot (1 - b_5) \cdot b_0 = 0 \text{ | degree} = 3
$$

Putting these operations into a group with flag degree 6 is important for two other reasons:
* Constraints for `U32SPLIT` operation have degree 3, and thus, degree of op flag for this operation cannot exceed 6.
* Operations `U32ADD3` and `U32MADD` shift the stack to the left. Thus, having these two operations in this group and putting them under the common prefix `10011`, allows us to create a common flag for these operations of degree 5 (recall that left-shift flag cannot exceed degree 5).

High-degree operations
This group contains operations which require constraints with degree up to 3. Similar to the previous group, the last op bit is not used in the computation of flag values for these operations.

Operation	Opcode value	Binary encoding	Operation group	Flag degree
`HPERM`	80	`101_0000`	[Crypto ops](./crypto_ops.md)	6
`MPVERIFY`	82	`101_0010`	[Crypto ops](./crypto_ops.md)	6
`PIPE`	84	`101_0100`	[I/O ops](./io_ops.md)	6
`MSTREAM`	86	`101_0110`	[I/O ops](./io_ops.md)	6
`SPAN`	88	`101_1000`	[Flow control ops](../decoder/main.md)	6
`JOIN`	90	`101_1010`	[Flow control ops](../decoder/main.md)	6
`SPLIT`	92	`101_1100`	[Flow control ops](../decoder/main.md)	6
`LOOP`	94	`101_1110`	[Flow control ops](../decoder/main.md)	6

Note that `SPLIT` and `LOOP` operations are grouped together under the common prefix `10111`, and thus, can have a common flag of degree 5. This is important because both of these operations shift the stack to the left.

Very high-degree operations
This group contains operations which require constraints with degree up to 5.

Operation	Opcode value	Binary encoding	Operation group	Flag degree
`MRUPDATE`	96	`11_00000`	[Crypto ops](./crypto_ops.md)	4
`PUSH`	100	`11_00100`	[I/O ops](./io_ops.md)	4
`SYSCALL`	104	`11_01000`	[Flow control ops](../decoder/main.md)	4
`CALL`	108	`11_01100`	[Flow control ops](../decoder/main.md)	4
`END`	112	`11_10000`	[Flow control ops](../decoder/main.md)	4
`REPEAT`	116	`11_10100`	[Flow control ops](../decoder/main.md)	4
`RESPAN`	120	`11_11000`	[Flow control ops](../decoder/main.md)	4
`HALT`	124	`11_11100`	[Flow control ops](../decoder/main.md)	4

As mentioned previously, the last two bits of the opcode are not used in computation of the flag for these operations. We force these bits to always be set to 0 with the following constraints:

$$
b_6 \cdot b_5 \cdot b_0 = 0 \text{ | degree} = 3
$$

$$
b_6 \cdot b_5 \cdot b_1 = 0 \text{ | degree} = 3
$$

Composite flags
Using the operation flags defined above, we can compute several composite flags which are used by various constraints in the VM.

Shift right flag
The right-shift flag indicates that an operation shifts the stack to the right. This flag is computed as follows:

$$
f_{shr} = (1 - b_6) \cdot b_5 \cdot b_4 + f_{u32split} + f_{push} \text{ | degree} = 6
$$

In the above, $(1 - b_6) \cdot b_5 \cdot b_4$ evaluates to 1 for all [right stack shift](#right-stack-shift-operations) operations described previously. This works because all these operations have a common prefix `011`. We also need to add in flags for other operations which shift the stack to the right but are not a part of the above group (e.g., `PUSH` operation).

Shift left flag
The left-shift flag indicates that a given operation shifts the stack to the left. To simplify the description of this flag, we will first compute the following intermediate variables:

A flag which is set to 1 when $f_{u32add3} = 1$ or $f_{u32madd} = 1$:

$$
f_{add3_madd} = b_6 \cdot (1 - b_5) \cdot (1 - b_4) \cdot b_3 \cdot b_2 \text{ | degree} = 5
$$

A flag which is set to 1 when $f_{split} = 1$ or $f_{loop} = 1$:

$$
f_{split_loop} = b_6 \cdot (1 - b_5) \cdot b_4 \cdot b_3 \cdot b_2 \text{ | degree} = 5
$$

Using the above variables, we compute left-shift flag as follows:

$$
f_{shl} = (1 - b_6) \cdot b_5 \cdot (1 - b_4) + f_{add3_madd} + f_{split_loop} + f_{repeat} + f_{end} \cdot h_5 \text{ | degree}
= 5
$$

In the above:
* $(1 - b_6) \cdot b_5 \cdot (1 - b_4)$ evaluates to 1 for all [left stack shift](#left-stack-shift-operations) operations described previously. This works because all these operations have a common prefix `010`.
* h_5 is the helper register in the decoder which is set to 1 when we are exiting a `LOOP` block, and to 0 otherwise.

Thus, similarly to the right-shift flag, we compute the value of the left-shift flag based on the prefix of the operation group which contains most left shift operations, and add in flag values for other operations which shift the stack to the left but are not a part of this group.

Control flow flag
The control flow flag f_{ctrl} is set to 1 when a control flow operation is being executed by the VM, and to 0 otherwise. Naively, this flag can be computed as follows:

$$
f_{ctrl} = f_{join} + f_{split} + f_{loop} + f_{repeat} + f_{span} + f_{respan} + f_{call} + f_{syscall} + f_{end} + f_{halt}
\text{ | degree} = 6
$$

However, this can be computed more efficiently via the common operation prefixes for the two groups of control flow operations as follows.

$$
f_{span,join,split,loop} = b_6 \cdot (1 - b_5) \cdot b_4 \cdot b_3 \text{ | degree} = 4
$$

$$
f_{end,repeat,respan,halt} = b_6 \cdot b_5 \cdot b_4 \text{ | degree} = 3
$$

$$
f_{ctrl} = f_{span,join,split,loop} + f_{end,repeat,respan,halt} + f_{call} + f_{syscall} \text{ | degree} = 4
$$

Note that the degree of $f_{end,repeat,respan,halt}$ can be reduced to degree 2 using the extra column, but this will not affect the degree of the f_{ctrl} constraint.
---
id: stack_ops
title: Stack Manipulation
sidebar_label: Stack Manipulation
description: "AIR constraints for Miden VM stack manipulation operations."
keywords:
  - docs
  - matic
  - polygon
  - wiki
  - miden
  - design
  - stack
image: https://wiki.polygon.technology/img/thumbnail/polygon-miden.png
---

In this section we describe the AIR constraints for Miden VM stack manipulation operations.

PAD
The `PAD` operation pushes a 0 onto the stack. The diagram below illustrates this graphically.

![pad](../../assets/design/stack/stack_ops/PAD.png)

Stack transition for this operation must satisfy the following constraints:

$$

```
s_{0}' = 0 \text{ | degree} = 1
$$

The effect of this operation on the rest of the stack is:
* **Right shift** starting from position $0$.

## DROP
The `DROP` operation removes an element from the top of the stack. The diagram below illustrates this graphically.

![drop](../../assets/design/stack/stack_ops/DROP.png)

The `DROP` operation shifts the stack by $1$ element to the left, but does not impose any additional constraints. The degree of left shift constraints is $1$.

The effect of this operation on the rest of the stack is:
* **Left shift** starting from position $1$.


## DUP(n)
The `DUP(n)` operations push a copy of the $n$-th stack element onto the stack. Eg. `DUP` (same as `DUP0`) pushes a copy of the top stack element onto the stack. Similarly, `DUP5` pushes a copy of the $6$-th stack element onto the stack. This operation is valid for $n \in \{0, ..., 7, 9, 11, 13, 15\}$. The diagram below illustrates this graphically.

![dupn](../../assets/design/stack/stack_ops/DUP(n).png)

Stack transition for this operation must satisfy the following constraints:

$$
s_{0}' - s_{n} = 0 \text{ for } n \in \{0, ..., 7, 9, 11, 13, 15\} \text{ | degree} = 1
$$

where $n$ is the depth of the stack from where the element has been copied.

The effect of this operation on the rest of the stack is:
* **Right shift** starting from position $0$.

## SWAP
The `SWAP` operations swaps the top two elements of the stack. The diagram below illustrates this graphically.

![swap](../../assets/design/stack/stack_ops/SWAP.png)

Stack transition for this operation must satisfy the following constraints:

$$
s_{0}' - s_{1} = 0 \text{ | degree} = 1
$$

$$
s_{1}' - s_{0} = 0 \text{ | degree} = 1
$$

The effect of this operation on the rest of the stack is:
* **No change** starting from position $2$.

## SWAPW
The `SWAPW` operation swaps stack elements $0, 1, 2, 3$ with elements $4, 5, 6, 7$. The diagram below illustrates this graphically.

![swapw](../../assets/design/stack/stack_ops/SWAPW.png)

Stack transition for this operation must satisfy the following constraints:

$$
s_{i}' - s_{i+4} = 0 \text{ for } i \in \{0, 1, 2, 3\} \text{ | degree} = 1
$$

$$
s_{i + 4}' - s_i = 0 \text{ for } i \in \{0, 1, 2, 3\} \text{ | degree} = 1
$$

The effect of this operation on the rest of the stack is:
* **No change** starting from position $8$.

## SWAPW2
The `SWAPW2` operation swaps stack elements $0, 1, 2, 3$ with elements $8, 9, 10, 11$. The diagram below illustrates this graphically.

![swapw2](../../assets/design/stack/stack_ops/SWAPW2.png)
```

Stack transition for this operation must satisfy the following constraints:

$$
s_i' - s_{i+8} = 0 \text{ for } i \in \{0, 1, 2, 3\} \text{ | degree} = 1
$$

$$
s_{i + 8}' - s_i = 0 \text{ for } i \in \{0, 1, 2, 3\} \text{ | degree} = 1
$$

The effect of this operation on the rest of the stack is:
* **No change** for elements $4, 5, 6, 7$.
* **No change** starting from position 12.

SWAPW3
The `SWAPW3` operation swaps stack elements $0, 1, 2, 3$ with elements $12, 13, 14, 15$. The diagram below illustrates this graphically.

![swapw3](../../assets/design/stack/stack_ops/SWAPW3.png)

Stack transition for this operation must satisfy the following constraints:

$$
s_i' - s_{i+12} = 0 \text{ for } i \in \{0, 1, 2, 3\} \text{ | degree} = 1
$$

$$
s_{i+12}' - s_i = 0 \text{ for } i \in \{0, 1, 2, 3\} \text{ | degree} = 1
$$

The effect of this operation on the rest of the stack is:
* **No change** for elements $4, 5, 6, 7, 8, 9, 10, 11$.
* **No change** starting from position 16.

SWAPDW
The `SWAPDW` operation swaps stack elements $\{0, ..., 7\}$ with elements $\{8, ..., 15\}$. The diagram below illustrates this graphically.

![swapdw](../../assets/design/stack/stack_ops/SWAPDW.png)

Stack transition for this operation must satisfy the following constraints:

$$
s_i' - s_{i+8} = 0 \text{ for } i \in \{0, ..., 7\}   \text{ | degree} = 1
$$

$$
s_{i+8}' - s_i = 0 \text{ for } i \in \{0, ..., 7\}   \text{ | degree} = 1
$$

The effect of this operation on the rest of the stack is:
* **No change** starting from position 16.

MOVUP(n)
The `MOVUP(n)` operation moves the n-th element of the stack to the top of the stack. For example, `MOVUP2` moves element at depth 2 to the top of the stack. All elements with depth less than n are shifted to the right by one, while elements with depth greater than n remain in place, and the depth of the stack does not change. This operation is valid for $n \in \{2, ..., 8\}$. The diagram below illustrates this graphically.

![movup](../../assets/design/stack/stack_ops/MOVUP(n).png)

Stack transition for this operation must satisfy the following constraints:

$$
s_0' - s_n = 0 \text{ for } n \in \{2, ..., 8\} \text{ | degree} = 1
$$

where n is the depth of the element which is moved moved to the top of the stack.

The effect of this operation on the rest of the stack is:
* **Right shift** for elements between 0 and $n-1$.
* **No change** starting from position $n+1$.

MOVDN(n)
The `MOVDN(n)` operation moves the top element of the stack to the n-th position. For example, `MOVDN2` moves the top element of the stack to depth 2. All the elements with depth less than n are shifted to the left by one, while elements with depth greater than n remain in place, and the depth of the stack does not change. This operation is valid for $n \in \{2, ..., 8\}$.

The diagram below illustrates this graphically.

![movdn](../../assets/design/stack/stack_ops/MOVDN(n).png)

Stack transition for this operation must satisfy the following constraints:

$$
s_n' - s_0 = 0 \text{ for } n \in \{2, ..., 8\} \text{ | degree} = 1
$$

where n is the depth to which the top stack element is moved.

The effect of this operation on the rest of the stack is:
* **Right shift** for elements between 0 and $n-1$.
* **No change** starting from position $n+1$.

CSWAP
The `CSWAP` operation pops an element off the stack and if the element is 1, swaps the top two remaining elements. If the popped element is 0, the rest of the stack remains unchanged. The diagram below illustrates this graphically.

![cswap](../../assets/design/stack/stack_ops/CSWAP.png)

In the above:

$$
d = \begin{cases} a, & \text{if}\ c = 0 \\ b, & \text{if}\ c = 1\ \end{cases} e = \begin{cases} b, & \text{if}\ c = 0 \\ a, & \text{if}\ c = 1\ \end{cases}
$$

Stack transition for this operation must satisfy the following constraints:

$$
s_0' - s_{0} \cdot s_{2} - (1-s_0) \cdot s_1 = 0 \text{ | degree} = 2
$$

$$
s_1' - s_0 \cdot s_{1} - (1-s_0) \cdot s_2 = 0 \text{ | degree} = 2
$$

We also need to enforce that the value in s_0 is binary. This can be done with the following constraint:

$$
s_0^2 - s_0 = 0 \text{ | degree} = 2
$$

The effect of this operation on the rest of the stack is:
* **Left shift** starting from position 3.

CSWAPW
The `CSWAPW` operation pops an element off the stack and if the element is 1, swaps elements $1, 2, 3, 4$ with elements $5, 6, 7, 8$. If the popped element is 0, the rest of the stack remains unchanged. The diagram below illustrates this graphically.

![cswapw](../../assets/design/stack/stack_ops/CSWAPW.png)

In the above:

$$
D = \begin{cases} A, & \text{if}\ c = 0 \\ B, & \text{if}\ c = 1\ \end{cases} E = \begin{cases} B, & \text{if}\ c = 0 \\ A, & \text{if}\ c = 1\ \end{cases}
$$

Stack transition for this operation must satisfy the following constraints:

$$
s_i' - s_0 \cdot s_{i+5} - (1-s_0) \cdot s_{i+1} = 0 \text{ for } i \in \{0..3\} \text{ | degree} = 2
$$

$$
s_{i+4}' - s_0 \cdot s_{i+1} + (1-s_0) \cdot s_{i+5} = 0 \text{ for } i \in \{0..3\} \text{ | degree} = 2
$$

We also need to enforce that the value in s_0 is binary. This can be done with the following constraint:

$$
s_0^2 - s_0 = 0 \text{ | degree} = 2
$$

The effect of this operation on the rest of the stack is:

```
* **Left shift** starting from position $9$.
---
id: system_ops
title:  System Operations
sidebar_label:  System Operations
description: "AIR constraints for Miden VM system operations."
keywords:
  - docs
  - matic
  - polygon
  - wiki
  - miden
  - design
  - system
  - operation
image: https://wiki.polygon.technology/img/thumbnail/polygon-miden.png
---

In this section we describe the AIR constraints for Miden VM system operations.

## NOOP
The `NOOP` operation advances the cycle counter but does not change the state of the operand stack (i.e., the depth of the stack
and the values on the stack remain the same).

The `NOOP` operation does not impose any constraints besides the ones needed to ensure that the entire state of the stack is
copied over. This constraint looks like so:

$$
s'_i - s_i = 0 \ \text{ for } i \in \{0, .., 15\} \text { | degree} = 1
$$

## ASSERT
The `ASSERT` operation pops an element off the stack and checks if the popped element is equal to $1$. If the element is not equal
to $1$, program execution fails..

![assert](../../assets/design/stack/system_ops/ASSERT.png)

Stack transition for this operation must satisfy the following constraints:

$$
s_0 - 1 = 0 \text{ | degree} = 1
$$

The effect on the rest of the stack is:
* **Left shift** starting from position $1$.

## FMPADD
The `FMPADD` operation pops an element off the stack, adds the current value of the `fmp` register to it, and pushes the result
back onto the stack. The diagram below illustrates this graphically.

![fmpadd](../../assets/design/stack/system_ops/FMPADD.png)

Stack transition for this operation must satisfy the following constraints:

$$
s_0' - (s_0 + fmp) = 0 \text{ | degree} = 1
$$

The effect on the rest of the stack is:
* **No change** starting from position $1$.

## FMPUPDATE
The `FMPUPDATE` operation pops an element off the stack and adds it to the current value of the `fmp` register. The diagram below
illustrates this graphically.

![fmpupdate](../../assets/design/stack/system_ops/FMPUPDATE.png)

The stack transition for this operation must follow the following constraint:

$$
fmp' - (fmp + s_0) = 0 \text{ | degree} = 1
$$

The effect on the rest of the stack is:
* **Left shift** starting from position $1$.

## CLK
The `CLK` operation pushes the current value of the clock cycle onto the stack. The diagram below illustrates this graphically.
```

```
![clk](../../assets/design/stack/system_ops/CLK.png)

The stack transition for this operation must follow the following constraint:

$$
s_0' - clk = 0 \text{ | degree} = 1
$$

The effect on the rest of the stack is:
* **Right shift** starting from position $0$.
---
id: u32_ops
title:  u32 Operations
sidebar_label:  u32 Operations
description: "Semantics and AIR constraints of operations over u32 values."
keywords:
  - docs
  - matic
  - polygon
  - wiki
  - miden
  - design
  - range
image: https://wiki.polygon.technology/img/thumbnail/polygon-miden.png
---

In this section we describe semantics and AIR constraints of operations over u32 values (i.e., 32-bit unsigned integers) as they
are implemented in Miden VM.

### Range checks
Most operations described below require some number of 16-bit range checks (i.e., verifying that the value of a field element is
smaller than $2^{16}$). The number of required range checks varies between $2$ and $4$, depending on the operation. However, to
simplify the constraint system, we force each relevant operation to consume exactly $4$ range checks.

To perform these range checks, the prover puts the values to be range-checked into helper registers $h_0, ..., h_3$, and then
divides the range checker bus column $b_{range}$ by a randomized product of these values. This operation is enforced via the
following constraint:

$$
b_{range}' \cdot (\alpha_0 + h_0) \cdot (\alpha_0 + h_1) \cdot (\alpha_0 + h_2) \cdot (\alpha_0 + h_3) = b_{range} \text{ |
degree} = 5
$$

The above is just a partial constraint as it does not show the range checker's part of the constraint, which multiplies the
required values into the bus column. It also omits the [selector flag](./op_constraints.md#operation-flags) which is used to turn
this constraint on only when executing relevant operations.

### Checking element validity
Another primitive which is required by most of the operations described below is checking whether four 16-bit values form a valid
field element. Assume $t_0$, $t_1$, $t_2$, and $t_3$ are known to be 16-bit values, and we want to verify that $2^{48} \cdot t_3 +
2^{32} \cdot t_2 + 2^{16} \cdot t_1 + t_0$ is a valid field element.

For simplicity, let's denote:

$$
v_{hi} = 2^{16} \cdot t_3 + t_2 \\
v_{lo} = 2^{16} \cdot t_1 + t_0
$$

We can then impose the following constraint to verify element validity:

$$
\left(1 - m \cdot (2^{32} - 1 - v_{hi})\right) \cdot v_{lo} = 0 \text{ | degree} = 3
$$

Where $m$ is a value set non-deterministically by the prover.

The above constraint should hold only if either of the following hold:

* $v_{lo} = 0$
* $v_{hi} \ne 2^{32} - 1$

To satisfy the latter equation, the prover needs to set $m = (2^{32} - 1 - v_{hi})^{-1}$, which is possible only when $v_{hi} \ne
2^{32} - 1$.

This constraint is sufficient because modulus $2^{64} - 2^{32} + 1$ in binary representation is 32 ones, followed by 31 zeros,
followed by a single one:
```

$$
1111111111111111111111111111111100000000000000000000000000000001
$$

This implies that the largest possible 64-bit value encoding a valid field element would be 32 ones, followed by 32 zeros:

$$
1111111111111111111111111111111100000000000000000000000000000000
$$

Thus, for a 64-bit value to encode a valid field element, either the lower 32 bits must be all zeros, or the upper 32 bits must not be all ones (which is $2^{32} - 1$).

U32SPLIT
Assume a is the element at the top of the stack. The `U32SPLIT` operation computes $(b,c) \leftarrow a$, where b contains the lower 32 bits of a, and c contains the upper 32 bits of a. The diagram below illustrates this graphically.

![u32split](../../assets/design/stack/u32_operations/U32SPLIT.png)

To facilitate this operation, the prover sets values in $h_0, ..., h_3$ to 16-bit limbs of a with h_0 being the least significant limb. Thus, stack transition for this operation must satisfy the following constraints:

$$
s_{0} = 2^{48} \cdot h_3 + 2^{32} \cdot h_2 + 2^{16} \cdot h_1 + h_0 \text{ | degree} = 1
$$

$$
s_{1}' = 2^{16} \cdot h_1 + h_0 \text{ | degree} = 1
$$

$$
s_{0}' = 2^{16} \cdot h_3 + h_2 \text{ | degree} = 1
$$

In addition to the above constraints, we also need to verify that values in $h_0, ..., h_3$ are smaller than 2^{16}, which we can do using 16-bit range checks as described [previously](#range-checks). Also, we need to make sure that values in $h_0, ..., h_3$, when combined, form a valid field element, which we can do by putting a nondeterministic value m into helper register h_4 and using the technique described [here](#checking-element-validity).

The effect of this operation on the rest of the stack is:
* **Right shift** starting from position 1.

U32ASSERT2
Assume a and b are the elements at the top of the stack. The `U32ASSERT2` verifies that both a and b are smaller than 2^{32}. The diagram below illustrates this graphically.

![u32assert2](../../assets/design/stack/u32_operations/U32ASSERT2.png)

To facilitate this operation, the prover sets values in h_0 and h_1 to low and high 16-bit limbs of a, and values in h_2 and h_3 to to low and high 16-bit limbs of b. Thus, stack transition for this operation must satisfy the following constraints:

$$
s_0' = 2^{16} \cdot h_3 + h_2 \text{ | degree} = 1
$$

$$
s_1' = 2^{16} \cdot h_1 + h_0 \text{ | degree} = 1
$$

In addition to the above constraints, we also need to verify that values in $h_0, ..., h_3$ are smaller than 2^{16}, which we can do using 16-bit range checks as described [previously](#range-checks).

The effect of this operation on the rest of the stack is:
* **No change** starting from position 0 - i.e., the state of the stack does not change.

U32ADD
Assume a and b are the values at the top of the stack which are known to be smaller than 2^{32}. The `U32ADD` operation computes $(c,d) \leftarrow a + b$, where c contains the low 32-bits of the result, and d is the carry bit. The diagram below illustrates this graphically.

![u32add](../../assets/design/stack/u32_operations/U32ADD.png)

To facilitate this operation, the prover sets values in h_0, h_1, and h_2 to 16-bit limbs of $a+b$ with h_0 being the least significant limb. Value in h_3 is set to 0. Thus, stack transition for this operation must satisfy the following constraints:

$$
s_0 + s_1 = 2^{32} \cdot h_2 + 2^{16} \cdot h_1 + h_0 \text{ | degree} = 1
$$

$$
s_0' = h_2 \text{ | degree} = 1
$$

$$
s_1' = 2^{16} \cdot h_1 + h_0 \text{ | degree} = 1
$$

In addition to the above constraints, we also need to verify that values in $h_0, ..., h_3$ are smaller than 2^{16}, which we can do using 16-bit range checks as described [previously](#range-checks).

The effect of this operation on the rest of the stack is:
* **No change** starting from position 2.

U32ADD3
Assume a, b, c are the values at the top of the stack which are known to be smaller than 2^{32}. The `U32ADD3` operation computes $(d, e) \leftarrow a + b + c$, where c and d contains the low and the high 32-bits of the result respectively. The diagram below illustrates this graphically.

![u32add3](../../assets/design/stack/u32_operations/U32ADD3.png)

To facilitate this operation, the prover sets values in h_0, h_1, and h_2 to 16-bit limbs of $a+b+c$ with h_0 being the least significant limb. Value in h_3 is set to 0. Thus, stack transition for this operation must satisfy the following constraints:

$$
s_0 + s_1 + s_2 = 2^{32} \cdot h_2 + 2^{16} \cdot h_1 + h_0 \text{ | degree} = 1
$$

$$
s_0' = h_2 \text{ | degree} = 1
$$

$$
s_1' = 2^{16} \cdot h_1 + h_0 \text{ | degree} = 1
$$

In addition to the above constraints, we also need to verify that values in $h_0, ..., h_3$ are smaller than 2^{16}, which we can do using 16-bit range checks as described [previously](#range-checks).

The effect of this operation on the rest of the stack is:
* **Left shift** starting from position 3.

U32SUB
Assume a and b are the values at the top of the stack which are known to be smaller than 2^{32}. The `U32SUB` operation computes $(c, d) \leftarrow a - b$, where c contains the 32-bit result in two's complement, and d is the borrow bit. The diagram below illustrates this graphically.

![u32sub](../../assets/design/stack/u32_operations/U32SUB.png)

To facilitate this operation, the prover sets values in h_0 and h_1 to the low and the high 16-bit limbs of $a-b$ respectively. Values in h_2 and h_3 are set to 0. Thus, stack transition for this operation must satisfy the following constraints:

$$
s_1 = s_0 + s_1' + 2^{32} \cdot s_0' \text{ | degree} = 1
$$

$$
s_0'^2 - s_0' = 0 \text{ | degree} = 2
$$

$$
s_1' = 2^{16} \cdot h_1 + h_0 \text{ | degree} = 1
$$

In addition to the above constraints, we also need to verify that values in $h_0, ..., h_3$ are smaller than 2^{16}, which we can do using 16-bit range checks as described [previously](#range-checks).

The effect of this operation on the rest of the stack is:
* **No change** starting from position 2.

U32MUL
Assume a and b are the values at the top of the stack which are known to be smaller than 2^{32}. The `U32MUL` operation

computes $(c, d) \leftarrow a \cdot b$, where c and d contain the low and the high 32-bits of the result respectively. The diagram below illustrates this graphically.

![u32mul](../../assets/design/stack/u32_operations/U32MUL.png)

To facilitate this operation, the prover sets values in $h_0, ..., h_3$ to 16-bit limbs of $a \cdot b$ with h_0 being the least significant limb. Thus, stack transition for this operation must satisfy the following constraints:

$$
s_0 \cdot s_1 = 2^{48} \cdot h_3 + 2^{32} \cdot h_2 + 2^{16} \cdot h_1 + h_0 \text{ | degree} = 2
$$

$$
s_1' = 2^{16} \cdot h_1 + h_0 \text{ | degree} = 1
$$

$$
s_0' = 2^{16} \cdot h_3 + h_2 \text{ | degree} = 1
$$

In addition to the above constraints, we also need to verify that values in $h_0, ..., h_3$ are smaller than 2^{16}, which we can do using 16-bit range checks as described [previously](#range-checks). Also, we need to make sure that values in $h_0, ..., h_3$, when combined, form a valid field element, which we can do by putting a nondeterministic value m into helper register h_4 and using the technique described [here](#checking-element-validity).

The effect of this operation on the rest of the stack is:
* **No change** starting from position 2.

U32MADD
Assume a, b, c are the values at the top of the stack which are known to be smaller than 2^{32}. The `U32MADD` operation computes $(d, e) \leftarrow a +b \cdot c$, where c and d contains the low and the high 32-bits of $a + b \cdot c$. The diagram below illustrates this graphically.

![u32madd](../../assets/design/stack/u32_operations/U32MADD.png)

To facilitate this operation, the prover sets values in $h_0, ..., h_3$ to 16-bit limbs of $a + b \cdot c$ with h_0 being the least significant limb. Thus, stack transition for this operation must satisfy the following constraints:

$$
s_0 \cdot s_1 + s_2 = 2^{48} \cdot h_3 + 2^{32} \cdot h_2 + 2^{16} \cdot h_1 + h_0 \text{ | degree} = 2
$$

$$
s_1' = 2^{16} \cdot h_1 + h_0 \text{ | degree} = 1
$$

$$
s_0' = 2^{16} \cdot h_3 + h_2 \text{ | degree} = 1
$$

In addition to the above constraints, we also need to verify that values in $h_0, ..., h_3$ are smaller than 2^{16}, which we can do using 16-bit range checks as described [previously](#range-checks). Also, we need to make sure that values in $h_0, ..., h_3$, when combined, form a valid field element, which we can do by putting a nondeterministic value m into helper register h_4 and using the technique described [here](#checking-element-validity).

Note: that the above constraints guarantee the correctness of the operation iff $a + b \cdot c$ cannot overflow field modules (which is the case for the field with modulus $2^{64} - 2^{32} + 1$).

The effect of this operation on the rest of the stack is:
* **Left shift** starting from position 3.

U32DIV
Assume a and b are the values at the top of the stack which are known to be smaller than 2^{32}. The `U32DIV` operation computes $(c, d) \leftarrow a / b$, where c contains the quotient and d contains the remainder. The diagram below illustrates this graphically.

![u32div](../../assets/design/stack/u32_operations/U32DIV.png)

To facilitate this operation, the prover sets values in h_0 and h_1 to 16-bit limbs of $a - c$, and values in h_2 and h_3 to 16-bit limbs of $b - d - 1$. Thus, stack transition for this operation must satisfy the following constraints:

$$
s_1 = s_0 \cdot s_1' + s_0' \text{ | degree} = 2
$$

$$
s_1 - s_1' = 2^{16} \cdot h_1 + h_0 \text{ | degree} = 1
$$

$$
s_0 - s_0' - 1= 2^{16} \cdot h_2 + h_3 \text{ | degree} = 1
$$

The second constraint enforces that $s_1' \leq s_1$, while the third constraint enforces that $s_0' < s_0$.

The effect of this operation on the rest of the stack is:
* **No change** starting from position 2.

U32AND
Assume a and b are the values at the top of the stack. The `U32AND` operation computes $c \leftarrow (a \land b)$, where c is the result of performing a bitwise AND on a and b. The diagram below illustrates this graphically.

![u32and](../../assets/design/stack/u32_operations/U32AND.png)

To facilitate this operation, we will need to make a request to the chiplet bus b_{chip} by dividing its current value by the value representing bitwise operation request. This can be enforced with the following constraint:

$$
b_{chip}' \cdot \left(\alpha_0 + \alpha_1 \cdot op_{u32and} + \alpha_2 \cdot s_0 + \alpha_3 \cdot s_1 + \alpha_4 \cdot s_0' \right) = b_{chip} \text{ | degree} = 2
$$

In the above, op_{u32and} is the unique [operation label](../chiplets/main.md#operation-labels) of the bitwise `AND` operation.

Note: unlike for many other u32 operations, bitwise AND operation does not assume that the values at the top of the stack are smaller than 2^{32}. This is because the lookup will fail for any inputs which are not 32-bit integers.

The effect of this operation on the rest of the stack is:
* **Left shift** starting from position 2.

U32XOR
Assume a and b are the values at the top of the stack. The `U32XOR` operation computes $c \leftarrow (a \oplus b)$, where c is the result of performing a bitwise XOR on a and b. The diagram below illustrates this graphically.

![u32xor](../../assets/design/stack/u32_operations/U32XOR.png)

To facilitate this operation, we will need to make a request to the chiplet bus b_{chip} by dividing its current value by the value representing bitwise operation request. This can be enforced with the following constraint:

$$
b_{chip}' \cdot \left(\alpha_0 + \alpha_1 \cdot op_{u32xor} + \alpha_2 \cdot s_0 + \alpha_3 \cdot s_1 + \alpha_4 \cdot s_0' \right) = b_{chip} \text{ | degree} = 2
$$

In the above, op_{u32xor} is the unique [operation label](../chiplets/main.md#operation-labels) of the bitwise `XOR` operation.

Note: unlike for many other u32 operations, bitwise XOR operation does not assume that the values at the top of the stack are smaller than 2^{32}. This is because the lookup will fail for any inputs which are not 32-bit integers.

The effect of this operation on the rest of the stack is:
* **Left shift** starting from position 2.
---
id: index
title: Welcome to Miden
hide_title: true
sidebar_label: Welcome
description: Miden Wiki is the documentation hub for Polygon Miden, providing extensive documentation, community resources, and guides for building on Miden VM.
keywords:
  - docs
  - Miden
  - wiki
  - polygon
image: https://wiki.polygon.technology/img/thumbnail/polygon-miden.png
---

import Welcome from "./landing.js";

<Welcome/>
---
id: architecture
title: Architecture of Miden VM
sidebar_label: Architecture
description: "Miden components and structure."
keywords:
  - docs

```
  - matic
  - polygon
  - miden
  - component
image: https://wiki.polygon.technology/img/thumbnail/polygon-miden.png
---

Miden VM is a stack machine. The base data type of the MV is a field element in a 64-bit [prime field]
(https://en.wikipedia.org/wiki/Finite_field) defined by modulus $p = 2^{64} - 2^{32} + 1$. This means that all values that the VM
operates with are field elements in this field (i.e., values between $0$ and $2^{64} - 2^{32}$, both inclusive).

Miden VM consists of three high-level components as illustrated below.

![](../assets/intro/vm_components.png)

These components are:
* **Stack** which is a push-down stack where each item is a field element. Most assembly instructions operate with values located
on the stack. The stack can grow up to $2^{32}$ items deep, however, only the top 16 items are directly accessible.
* **Memory** which is a linear random-access read-write memory. The memory is word-addressable, meaning, four elements are located
at each address, and we can read and write elements to/from memory in batches of four. Memory addresses can be in the range $[0,
2^{32})$.
* **Advice provider** which is a way for the prover to provide nondeterministic inputs to the VM. The advice provider is composed
of a single *advice stack*, an *advice map*, and a *merkle store*. The *advice stack* yields elements to the VM stack; the *advice
map* stores key-mapped element lists which can be pushed onto the advice stack; finally, the Merkle store contains structured
Merkle tree data and serves Merkle paths to the VM.

In the future, additional components (e.g., storage, logs) may be added to the VM.

## Writing programs
Our goal is to make Miden VM an easy compilation target for high-level blockchain-centric languages such as Solidity, Move, Sway,
and others. We believe it is important to let people write programs in the languages of their choice. However, compilers to help
with this have not been developed yet. Thus, for now, the primary way to write programs for Miden VM is to use [Miden assembly]
(../user_docs/assembly/main.md).

While writing programs in assembly is far from ideal, Miden assembly does make this task a little bit easier by supporting high-
level flow control structures and named procedures.

## Inputs and outputs
External inputs can be provided to Miden VM in two ways:

1. Public inputs can be supplied to the VM by initializing the stack with desired values before a program starts executing. Any
number of stack items can be initialized in this way, but providing a large number of public inputs will increase the cost for the
verifier.
2. Secret (or nondeterministic) inputs can be supplied to the VM via the [*advice provider*](#nondeterministic-inputs). There is
no limit on how much data the advice provider can hold.

After a program finishes executing, the elements remaining on the stack become the outputs of the program. Since these outputs
will be public inputs for the verifier, having a large stack at the end of execution will increase cost to the verifier.
Therefore, it's best to drop unneeded output values. We've provided the [`truncate_stack`](../user_docs/stdlib/sys.md) utility
function in the standard library for this purpose.

The number of public inputs and outputs of a program can be reduced by making use of the advice stack and Merkle trees. Just 4
elements are sufficient to represent a root of a Merkle tree, which can be expanded into an arbitrary number of values.

For example, if we wanted to provide a thousand public input values to the VM, we could put these values into a Merkle tree,
initialize the stack with the root of this tree, initialize the advice provider with the tree itself, and then retrieve values
from the tree during program execution using `mtree_get` instruction (described [here]
(../user_docs/assembly/cryptographic_operations.md#hashing-and-merkle-trees)).

In the future, other ways of providing public inputs and reading public outputs (e.g., storage commitments) may be added to the
VM.

### Stack depth restrictions
For reasons explained [here](../design/stack/main.md), the VM imposes the restriction that the stack depth cannot be smaller than
$16$. This has the following effects:

- When initializing a program with fewer than $16$ inputs, the VM will pad the stack with zeros to ensure the depth is $16$ at the
beginning of execution.
- If an operation would result in the stack depth dropping below $16$, the VM will insert a zero at the deep end of the stack to
make sure the depth stays at $16$.

### Nondeterministic inputs
The *advice provider* component is responsible for supplying nondeterministic inputs to the VM. These inputs only need to be known
to the prover (i.e., they do not need to be shared with the verifier).

The advice provider consists of three components:
* **Advice stack** which is a one-dimensional array of field elements. Being a stack, the VM can either push new elements onto the
advice stack, or pop the elements from its top.
```

* **Advice map** which is a key-value map where keys are words and values are vectors of field elements. The VM can copy values from the advice map onto the advice stack as well as insert new values into the advice map (e.g., from a region of memory).
* **Merkle store** which contain structured data reducible to Merkle paths. Some examples of such structures are: Merkle tree, Sparse Merkle tree, and a collection of Merkle paths. The VM can request Merkle paths from the Merkle store, as well as mutate it by updating or merging nodes contained in the store.

The prover initializes the advice provider prior to executing a program, and from that point on the advice provider is manipulated solely by executing operations on the VM.

```
---
id: main
title: Introduction to the Miden Virtual Machine
sidebar_label: Introduction
description: Status and features of the Miden virtual machine.
keywords:
  - docs
  - matic
  - polygon
  - miden
  - features
image: https://wiki.polygon.technology/img/thumbnail/polygon-miden.png
---
```

Miden VM is a zero-knowledge virtual machine written in Rust. For any program executed on Miden VM, a STARK-based proof of execution is automatically generated. This proof can then be used by anyone to verify that the program was executed correctly without the need for re-executing the program or even knowing the contents of the program.

Status and features
Miden VM is currently on release v0.5. In this release, most of the core features of the VM have been stabilized, and most of the STARK proof generation has been implemented. While we expect to keep making changes to the VM internals, the external interfaces should remain relatively stable, and we will do our best to minimize the amount of breaking changes going forward.

At this point, Miden VM is good enough for experimentation, and even for real-world applications, but it is not yet ready for production use. The codebase has not been audited and contains known and unknown bugs and security flaws.

Feature highlights
Miden VM is a fully-featured virtual machine. Despite being optimized for zero-knowledge proof generation, it provides all the features one would expect from a regular VM. To highlight a few:

* **Flow control.** Miden VM is Turing-complete and supports familiar flow control structures such as conditional statements and counter/condition-controlled loops. There are no restrictions on the maximum number of loop iterations or the depth of control flow logic.
* **Procedures.** Miden assembly programs can be broken into subroutines called *procedures*. This improves code modularity and helps reduce the size of Miden VM programs.
* **Execution contexts.** Miden VM program execution can span multiple isolated contexts, each with its own dedicated memory space. The contexts are separated into the *root context* and *user contexts*. The root context can be accessed from user contexts via customizable kernel calls.
* **Memory.** Miden VM supports read-write random-access memory. Procedures can reserve portions of global memory for easier management of local variables.
* **u32 operations.** Miden VM supports native operations with 32-bit unsigned integers. This includes basic arithmetic, comparison, and bitwise operations.
* **Cryptographic operations.** Miden assembly provides built-in instructions for computing hashes and verifying Merkle paths. These instructions use Rescue Prime Optimized hash function (which is the native hash function of the VM).
* **External libraries.** Miden VM supports compiling programs against pre-defined libraries. The VM ships with one such library: Miden `stdlib` which adds support for such things as 64-bit unsigned integers. Developers can build other similar libraries to extend the VM's functionality in ways which fit their use cases.
* **Nondeterminism**. Unlike traditional virtual machines, Miden VM supports nondeterministic programming. This means a prover may do additional work outside of the VM and then provide execution *hints* to the VM. These hints can be used to dramatically speed up certain types of computations, as well as to supply secret inputs to the VM.
* **Custom advice providers.** Miden VM can be instantiated with user-defined advice providers. These advice providers are used to supply external data to the VM during execution/proof generation (via nondeterministic inputs) and can connect the VM to arbitrary data sources (e.g., a database or RPC calls).

Planned Features

In the coming months we plan to finalize the design of the VM and implement support for the following features:

* **Recursive proofs.** Miden VM will soon be able to verify a proof of its own execution. This will enable infinitely recursive proofs, an extremely useful tool for real-world applications.
* **Better debugging.** Miden VM will provide a better debugging experience including the ability to place breakpoints, better source mapping, and more complete program analysis info.
* **Faulty execution.** Miden VM will support generating proofs for programs with faulty execution (a notoriously complex task in ZK context). That is, it will be possible to prove that execution of some program resulted in an error.

Documentatioin Structure

This document is meant to provide an in-depth description of Miden VM. It is organized as follows:

* In the introduction, we provide a high-level overview of Miden VM and describe how to run simple programs.

* In the user documentation section, we provide developer-focused documentation useful to those who want to develop on Miden VM or build compilers from higher-level languages to Miden assembly (the native language of Miden VM).
* In the design section, we provide in-depth descriptions of the VM's internals, including all AIR constraints for the proving system. We also provide the rationale for settling on specific design choices.
* Finally, in the background material section, we provide references to materials which could be useful for learning more about STARKs - the proving system behind Miden VM.

In the reference section, we provide developer-focused documentation useful to those who want to develop on Miden VM or build compilers from higher-level languages to Miden VM.

This documentation consists of two high-level sections:

- [Miden assembly](./assembly/main.md) which provides a detailed description of Miden assembly language, which is the native language of Miden VM.
- [Miden Standard Library](./stdlib/main.md) which provides descriptions of all procedures available in Miden Standard Library.

For info on how to run programs on Miden VM, please refer to the [usage](/docs/miden/intro/usage.md) section in the introduction.
---
id: performance
title: Performance
sidebar_label: Performance
description: "Performance benchmarks."
keywords:
  - docs
  - matic
  - polygon
  - miden
  - performance
  - benchmark
image: https://wiki.polygon.technology/img/thumbnail/polygon-miden.png
---

The benchmarks below should be viewed only as a rough guide for expected future performance. The reasons for this are twofold:
1. Not all constraints have been implemented yet, and we expect that there will be some slowdown once constraint evaluation is completed.
2. Many optimizations have not been applied yet, and we expect that there will be some speedup once we dedicate some time to performance optimizations.

Overall, we don't expect the benchmarks to change significantly, but there will definitely be some deviation from the below numbers in the future.

A few general notes on performance:

* Execution time is dominated by proof generation time. In fact, the time needed to run the program is usually under 1% of the time needed to generate the proof.
* Proof verification time is really fast. In most cases it is under 1 ms, but sometimes gets as high as 2 ms or 3 ms.
* Proof generation process is dynamically adjustable. In general, there is a trade-off between execution time, proof size, and security level (i.e. for a given security level, we can reduce proof size by increasing execution time, up to a point).
* Both proof generation and proof verification times are greatly influenced by the hash function used in the STARK protocol. In the benchmarks below, we use BLAKE3, which is a really fast hash function.

Single-core prover performance
When executed on a single CPU core, the current version of Miden VM operates at around 10 - 15 KHz. In the benchmarks below, the VM executes a Fibonacci calculator program on Apple M1 Pro CPU in a single thread. The generated proofs have a target security level of 96 bits.

VM cycles	Execution time	Proving time	RAM consumed	Proof size
2¹⁰	1 ms	80 ms	20 MB	47 KB
2¹²	2 ms	260 ms	52 MB	57 KB
2¹⁴	8 ms	0.9 sec	240 MB	66 KB
2¹⁶	28 ms	4.6 sec	950 MB	77 KB
2¹⁸	85 ms	15.5 sec	3.7 GB	89 KB
2²⁰	310 ms	67 sec	14 GB	100 KB

As can be seen from the above, proving time roughly doubles with every doubling in the number of cycles, but proof size grows much slower.

We can also generate proofs at a higher security level. The cost of doing so is roughly doubling of proving time and roughly 40% increase in proof size. In the benchmarks below, the same Fibonacci calculator program was executed on Apple M1 Pro CPU at 128-bit target security level:

VM cycles	Execution time	Proving time	RAM consumed	Proof size
2¹⁰	1 ms	300 ms	30 MB	61 KB
2¹²	2 ms	590 ms	106 MB	78 KB
2¹⁴	8 ms	1.7 sec	500 MB	91 KB
2¹⁶	28 ms	6.7 sec	2.0 GB	106 KB

| 2¹⁸ | 85 ms | 27.5 sec | 8.0 GB | 122 KB |
| 2²⁰ | 310 ms | 126 sec | 24.0 GB | 138 KB |

Multi-core prover performance

STARK proof generation is massively parallelizable. Thus, by taking advantage of multiple CPU cores we can dramatically reduce proof generation time. For example, when executed on an 8-core CPU (Apple M1 Pro), the current version of Miden VM operates at around 100 KHz. And when executed on a 64-core CPU (Amazon Graviton 3), the VM operates at around 250 KHz.

In the benchmarks below, the VM executes the same Fibonacci calculator program for 2²⁰ cycles at 96-bit target security level:

Machine	Execution time	Proving time	Execution %
Apple M1 Pro (8 threads)	310 ms	9.8 sec	3.1%
Apple M2 Max (16 threads)	290 ms	7.7 sec	3.6%
AMD Ryzen 9 5950X (16 threads)	270 ms	10.7 sec	2.6%
Amazon Graviton 3 (64 threads)	330 ms	3.7 sec	9.0%

---
id: resources
title: Useful Resources
sidebar_label: Resources
description: Useful resources for exploring Polygon Miden.
keywords:
  - docs
  - matic
  - polygon
  - miden
  - polygon id
  - decentralized identity
image: https://wiki.polygon.technology/img/thumbnail/polygon-miden.png
---

Proofs of execution generated by Miden VM are based on STARKs. A STARK is a novel proof-of-computation scheme that allows you to create an efficiently verifiable proof that a computation was executed correctly. The scheme was developed by Eli Ben-Sasson, Michael Riabzev et al. at Technion - Israel Institute of Technology. STARKs do not require an initial trusted setup, and rely on very few cryptographic assumptions.

Here are some resources to learn more about STARKs:

* STARKs paper: [Scalable, transparent, and post-quantum secure computational integrity](https://eprint.iacr.org/2018/046)
* STARKs vs. SNARKs: [A Cambrian Explosion of Crypto Proofs](https://nakamoto.com/cambrian-explosion-of-crypto-proofs/)

Vitalik Buterin's blog series on zk-STARKs:
* [STARKs, part 1: Proofs with Polynomials](https://vitalik.ca/general/2017/11/09/starks_part_1.html)
* [STARKs, part 2: Thank Goodness it's FRI-day](https://vitalik.ca/general/2017/11/22/starks_part_2.html)
* [STARKs, part 3: Into the Weeds](https://vitalik.ca/general/2018/07/21/starks_part_3.html)

Alan Szepieniec's STARK tutorials:
* [Anatomy of a STARK](https://aszepieniec.github.io/stark-anatomy/)
* [BrainSTARK](https://aszepieniec.github.io/stark-brainfuck/)

StarkWare's STARK Math blog series:
* [STARK Math: The Journey Begins](https://medium.com/starkware/stark-math-the-journey-begins-51bd2b063c71)
* [Arithmetization I](https://medium.com/starkware/arithmetization-i-15c046390862)
* [Arithmetization II](https://medium.com/starkware/arithmetization-ii-403c3b3f4355)
* [Low Degree Testing](https://medium.com/starkware/low-degree-testing-f7614f5172db)
* [A Framework for Efficient STARKs](https://medium.com/starkware/a-framework-for-efficient-starks-19608ba06fbe)

StarkWare's STARK tutorial:
* [STARK 101](https://starkware.co/stark-101/)---
id: code_organization
title: Code Organization
sidebar_label: Code Organization
description: "Miden assembly's code structure."
keywords:
  - docs
  - matic
  - polygon
  - miden
  - assembly
image: https://wiki.polygon.technology/img/thumbnail/polygon-miden.png
---

A Miden assembly program is just a sequence of instructions each describing a specific directive or an operation. You can use any combination of whitespace characters to separate one instruction from another.

In turn, Miden assembly instructions are just keywords which can be parameterized by zero or more parameters. The notation for specifying parameters is *keyword.param1.param2* - i.e., the parameters are separated by periods. For example, `push.123`

```
instruction denotes a `push` operation which is parameterized by value `123`.

Miden assembly programs are organized into procedures. Procedures, in turn, can be grouped into modules.

### Procedures
A *procedure* can be used to encapsulate a frequently-used sequence of instructions which can later be invoked via a label. A procedure must start with a `proc.<label>.<number of locals>` instruction and terminate with an `end` instruction. For example:
```

proc.foo.2 end

```
A procedure label must start with a letter and can contain any combination of numbers, ASCII letters, and underscores (`_`). The number of characters in the procedure label cannot exceed 100.

The number of locals specifies the number of memory-based local words a procedure can access (via `loc_load`, `loc_store`, and [other instructions](./io_operations.md#random-access-memory)). If a procedure doesn't need any memory-based locals, this parameter can be omitted or set to `0`. A procedure can have at most $2^{16}$ locals, and the total number of locals available to all procedures at runtime is limited to $2^{30}$.

To execute a procedure, the `exec.<label>`, `call.<label>`, and `syscall.<label>` instructions can be used. For example:
```

exec.foo

```
The difference between using each of these instructions is explained in the [next section](./execution_contexts.md#procedure-invocation-semantics).

A procedure may execute any other previously defined procedure, but it cannot execute itself or any of the subsequent procedures. Thus, recursive procedure calls are not possible. For example, the following code block defines a program with two procedures:
```

proc.foo end

proc.bar exec.foo end

begin exec.bar exec.foo end

```
### Modules
A *module* consists of one or more procedures. There are two types of modules: *library modules* and *executable modules* (also called *programs*).

#### Library modules
Library modules contain zero or more internal procedures and one or more exported procedures. For example, the following module defines one internal procedure (defined with `proc` instruction) and one exported procedure (defined with `export` instruction):
```

proc.foo end

export.bar exec.foo end

```
#### Programs
Executable modules are used to define programs. A program contains zero or more internal procedures (defined with `proc` instruction) and exactly one main procedure (defined with `begin` instruction). For example, the following module defines one internal procedure and a main procedure:
```

proc.foo end

begin exec.foo end

```
A program cannot contain any exported procedures.

When a program is executed, the execution starts at the first instruction following the `begin` instruction. The main procedure is expected to be the last procedure in the program and can be followed only by comments.

#### Importing modules
To invoke a procedure from an external module, the module first needs to be imported using a `use` instruction. Once a module is imported, procedures from this module can be invoked via the regular `exec` or `call` instructions as `exec|call.<module>::<label>` where `label` is the name of the procedure. For example:
```

use.std::math::u64

begin push.1.0 push.2.0 exec.u64::checked_add end

```
In the above example we import `std::math::u64` module from the [standard library](../stdlib/main.md). We then execute a program which pushes two 64-bit integers onto the stack, and then invokes a 64-bit addition procedure from the imported module.

The set of modules which can be imported by a program can be specified via a Module Provider when instantiating the [Miden Assembler](https://crates.io/crates/miden-assembly) used to compile the program.

### Constants
Miden assembly supports constant declarations. These constants are scoped to the module they are defined in and can be used as immediate parameters for Miden assembly instructions. Currently only `push` instruction supports this.
```

```
Constants must be declared right after module imports and before any procedures or program bodies. A constant's name must start
with an upper-case letter and can contain any combination of numbers, upper-case ASCII letters, and underscores (`_`). The number
of characters in a constant name cannot exceed 100.
```

use.std::math::u64

const.CONSTANT_1=100 const.CONSTANT_2=200

begin push.CONSTANT_1.CONSTANT_2 exec.u64::checked_add end

```
### Comments
Miden assembly allows annotating code with simple comments. There are two types of comments: single-line comments which start with
a `#` (pound) character, and documentation comments which start with `#!` characters. For example:
```

#! This is a documentation comment export.foo # this is a comment push.1 end ``` Documentation comments must precede a procedure declaration. Using them inside a procedure body is an error.

id: cryptographic_operations title: Cryptographic Operations sidebar_label: Cryptographic Operations description: "Instructions for performing common cryptographic operations." keywords: - docs - matic - polygon - miden - cryptographic image: https://wiki.polygon.technology/img/thumbnail/polygon-miden.png

Miden assembly provides a set of instructions for performing common cryptographic operations. These instructions are listed in the table below.

Hashing and Merkle trees

Rescue Prime Optimized is the native hash function of Miden VM. The parameters of the hash function were chosen to provide 128-bit security level against preimage and collision attacks. The function operates over a state of 12 field elements, and requires 7 rounds for a single permutation. However, due to its special status within the VM, computing Rescue Prime Optimized hashes can be done very efficiently. For example, applying a permutation of the hash function can be done in a single VM cycle.

Instruction	Stack_input	Stack_output	Notes
hash *- (20 cycles)*	[A, ...]	[B, ...]	${B} \leftarrow hash(A)$ where, $hash()$ computes a 1-to-1 Rescue Prime Optimized hash.
hperm *- (1 cycle)*	[C, B, A, ...]	[F, E, D, ...]	${D, E, F} \leftarrow permute(A, B, C)$ where, $permute()$ computes a Rescue Prime Optimized permutation.
hmerge *- (16 cycles)*	[B, A, ...]	[C, ...]	$C \leftarrow hash(A,B)$ where, $hash()$ computes a 2-to-1 Rescue Prime Optimized hash.
mtree_get *- (9 cycles)*	[d, i, R, ...]	[V, R, ...]	Verifies that a Merkle tree with root R opens to node V at depth d and index i. Merkle tree with root R must be present in the advice provider, otherwise execution fails.
mtree_set *- (29 cycles)*	[d, i, R, V', ...]	[V, R', ...]	Updates a node in the Merkle tree with root R at depth d and index i to value V'. R' is the Merkle root of the resulting tree and V is old value of the node. Merkle tree with root R must be present in the advice provider, otherwise execution fails. At the end of the operation the advice provider will contain both Merkle trees.
mtree_merge *- (16 cycles)*	[R, L, ...]	[M, ...]	Merges two Merkle trees with the provided roots R (right), L (left) into a new Merkle tree with root M (merged). The input trees are retained in the advice provider.

---

id: execution-context title: Execution Contexts sidebar_label: Execution Contexts description: An execution context defines its own memory space which is not accessible from other execution contexts. keywords: - docs - wiki - polygon - miden - execution context image: https://wiki.polygon.technology/img/thumbnail/polygon-miden.png

Miden assembly program execution can span multiple isolated contexts. An execution context defines its own memory space which is not accessible from other execution contexts.

All programs start executing in a *root* context. Thus, the main procedure of a program is always executed in the root context. To move execution into a different context, we can invoke a procedure using the `call` instruction. In fact, any time we invoke a procedure using the `call` instruction, the procedure is executed in a new context. We refer to all non-root contexts as *user contexts*.

While executing in a user context, we can request to execute some procedures in the root context. This can be done via the `syscall` instruction. The set of procedures which can be invoked via the `syscall` instruction is limited by the kernel against which a program is compiled. Once the procedure called via `syscall` returns, the execution moves back to the user context from which it was invoked. The diagram below illustrates this graphically:

context transitions

Procedure invocation semantics

As mentioned in the previous section, procedures in Miden assembly can be invoked via three different instructions: `exec`, `call`, and `syscall`. Invocation semantics of `call` and `syscall` instructions are basically the same, the only difference being that the `syscall` instruction can be used only with procedures which are defined in the program's kernel. The `exec` instruction is different, and we explain these differences below.

Invoking via `call` and `syscall` instructions

When a procedure is invoked via a `call` or a `syscall` instruction, the following happens:

- Execution moves into a different context. In case of a `call` instruction, a new user context is created. In case of a `syscall` instruction, the execution moves back into the root context.
- All stack items beyond the 16th item get "hidden" from the invoked procedure. That is, from the standpoint of the invoked procedure, the initial stack depth is set to 16.

When a procedure returns from a `call` or a `syscall`, the following happens:

- Execution moves back to the context from which the procedure was invoked.
- Stack depth is set to its original depth. Before the stack depth is reset, the VM checks if the current stack depth is exactly 16, and fails otherwise.

The manipulations of the stack depth described above have the following implications:

- The top 16 elements of the stack can be used to pass parameters and return values between the caller and the callee.
- Caller's stack beyond the top 16 elements is inaccessible to the callee, and thus, is guaranteed not to change as the result of the call.
- At the end of its execution, the callee must ensure that stack depth is exactly 16. If this is difficult to ensure manually, the `truncate_stack` procedure can be used to drop all elements from the stack except for the top 16.

Invoking via `exec` instruction

Procedures invoked via the `exec` instruction, are inlined at their call sites during compilation. Thus, from the standpoint of the final program, executing procedures this way is indistinguishable from manually including procedure code in place of the `exec` instruction. This also means that procedures invoked via the `exec` instruction are executed in the same context as the caller.

Kernels

A *kernel* defines a set of procedures which can be invoked from user contexts to be executed in the root context. Miden assembly programs are always compiled against some kernel. The default kernel is empty - i.e., it does not contain any procedures. To compile a program against a non-empty kernel, the kernel needs to be specified when instantiating the Miden Assembler.

A kernel can be defined similarly to a regular library module - i.e., it can have internal and exported procedures. However, there are some small differences between what procedures can do in a kernel module vs. what they can do in a regular library module. Specifically:

- Procedures in a kernel module cannot use `call` or `syscall` instructions. This means that creating a new context from within a `syscall` is not possible.
- Unlike procedures in regular library modules, procedures in a kernel module can use the `caller` instruction. This instruction puts the hash of the procedure which initiated the parent context onto the stack.

Memory layout

As mentioned earlier, procedures executed within a given context can access memory only of that context. This is true for both memory reads and memory writes.

Address space of every context is the same: the smallest accessible address is 0 and the largest accessible address is $2^{32} - 1$. Any code executed in a given context has access to its entire address space. However, by convention, we assign different meanings to different regions of the address space.

For user contexts we have the following:

- The first 2^{30} words (each word is 4 field elements) are assumed to be global memory.
- The next 2^{30} words are reserved for memory locals of procedures executed in the same context (i.e., via the `exec` instruction).
- The remaining address space has no special meaning.

user memory layout

For the root context we have the following:

- The first 2^{30} words are assumed to be global memory.
- The next 2^{30} words are reserved for memory locals of procedures executed in the root context.
- The next 2^{30} words are reserved for memory locals of procedures executed from within a `syscall`.
- The remaining address space has no special meaning.

root memory layout

For both types of contexts, writing directly into regions of memory reserved for procedure locals is not advisable. Instead, `loc_load`, `loc_store` and other similar dedicated instructions should be used to access procedure locals.

Example

To better illustrate what happens as we execute procedures in different contexts, let's go over the following example.

```
kernel
--------------------
export.baz.2
    <instructions>
    caller
    <instructions>
end

program
--------------------
proc.bar.1
    <instructions>
    syscall.baz
    <instructions>
end

proc.foo.3
    <instructions>
    call.bar
    <instructions>
    exec.bar
    <instructions>
end

begin
    <instructions>
    call.foo
    <instructions>
end
```

Execution of the above program proceeds as follows: 1. The VM starts executing instructions immediately following the `begin` statement. These instructions are executed in the *root* context (let's call this context `ctx0`). 2. When `call.foo` is executed, a new context is created (`ctx1`). Memory in this context is isolated from `ctx0`. Additionally, any elements on the stack beyond the top 16 are hidden from `foo`. 3. Instructions executed inside `foo` can access memory of `ctx1` only. The address of the first procedure local in `foo` (e.g., accessed via `loc_load.0`) is 2^{30}. 4. When `call.bar` is executed, a new context is created (`ctx2`). The stack depth is set to 16 again, and any instruction executed in this context can access memory of `ctx2` only. The first procedure local of `bar` is also located at address 2^{30}. 5. When `syscall.baz` is executed, the execution moves back into the root context. That is, instructions executed inside `baz` have access to the memory of `ctx0`. The first procedure local of `baz` is located at address 2^{31}. When `baz` starts executing, the stack depth is again set to 16. 6. When `caller` is executed inside `baz`, the first 4 elements of the stack are populated with the hash of `bar` since `baz` was invoked from `bar`'s context. 7. Once `baz` returns, execution moves back to `ctx2`, and then, when `bar` returns, execution moves back to `ctx1`. We assume that instructions executed right before each procedure returns ensure that the stack depth is exactly 16 right before procedure's end. 8. Next, when `exec.bar` is executed, `bar` is executed again, but this time it is executed in the same context as `foo`. Thus, it can access memory of `ctx1`. Moreover, the stack depth is not changed, and thus, `bar` can access the entire stack of `foo`. Lastly, this first procedure local of `bar` now will be at address $2^{30} + 3$ (since the first 3 locals in this context are reserved for `foo`). 9. When `syscall.baz` is executed the second time, execution moves into the root context again. However, now, when `caller` is executed inside `baz`, the first 4 elements of the stack are populated with the hash of `foo` (not `bar`). This happens because this time around `bar` does not have its own context and `baz` is invoked from `foo`'s context. 10. Finally, when `baz` returns, execution moves back to `ctx1`, and then as `bar` and `foo` return, back to `ctx0`, and the program terminates.

id: field_operations title: Field Operations sidebar_label: Field Operations description: "Tables of operations with raw field elements." keywords: - docs - matic - polygon - miden - operation image: https://wiki.polygon.technology/img/thumbnail/polygon-miden.png

Miden assembly provides a set of instructions which can perform operations with raw field elements. These instructions are described in the tables below.

While most operations place no restrictions on inputs, some operations expect inputs to be binary values, and fail if executed with non-binary inputs.

For instructions where one or more operands can be provided as immediate parameters (e.g., `add` and `add.b`), we provide stack transition diagrams only for the non-immediate version. For the immediate version, it can be assumed that the operand with the specified name is not present on the stack.

Assertions and tests

Instruction	Stack_input	Stack_output	Notes
assert *- (1 cycle)*	[a, ...]	[...]	If $a = 1$, removes it from the stack. Fails if $a \ne 1$
assertz *- (2 cycles)*	[a, ...]	[...]	If $a = 0$, removes it from the stack, Fails if $a \ne 0$
assert_eq *- (2 cycles)*	[b, a, ...]	[...]	If $a = b$, removes them from the stack. Fails if $a \ne b$
assert_eqw *- (11 cycles)*	[B, A, ...]	[...]	If $A = B$, removes them from the stack. Fails if $A \ne B$

Arithmetic and Boolean operations

Instruction	Stack_input	Stack_output	Notes
add *- (1 cycle)* add.*b* *- (1-2 cycle)*	[b, a, ...]	[c, ...]	$c \leftarrow (a + b) \mod p$
sub *- (2 cycles)* sub.*b* *- (2 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow (a - b) \mod p$
mul *- (1 cycle)* mul.*b* *- (2 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow (a \cdot b) \mod p$
div *- (2 cycles)* div.*b* *- (2 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow (a \cdot b^{-1}) \mod p$ Fails if $b = 0$
neg *- (1 cycle)*	[a, ...]	[b, ...]	$b \leftarrow -a \mod p$
inv *- (1 cycle)*	[a, ...]	[b, ...]	$b \leftarrow a^{-1} \mod p$ Fails if $a = 0$
pow2 *- (16 cycles)*	[a, ...]	[b, ...]	$b \leftarrow 2^a$ Fails if $a > 63$
exp.*uxx* *- (9 + xx cycles)* exp.*b* *- (9 + log2(b) cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow a^b$ Fails if xx is outside [0, 63) exp is equivalent to exp.u64 and needs 73 cycles

not *- (1 cycle)*	[a, ...]	[b, ...]	$b \leftarrow 1 - a$ Fails if $a > 1$
and *- (1 cycle)*	[b, a, ...]	[c, ...]	$c \leftarrow a \cdot b$ Fails if $max(a, b) > 1$
or *- (1 cycle)*	[b, a, ...]	[c, ...]	$c \leftarrow a + b - a \cdot b$ Fails if $max(a, b) > 1$
xor *- (7 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow a + b - 2 \cdot a \cdot b$ Fails if $max(a, b) > 1$

Comparison operations

Instruction	Stack_input	Stack_output	Notes
eq *- (1 cycle)* eq.*b* *- (1-2 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow \begin{cases} 1, & \text{if}\ a=b \ 0, & \text{otherwise}\ \end{cases}$
neq *- (2 cycle)* neq.*b* *- (2-3 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow \begin{cases} 1, & \text{if}\ a \ne b \ 0, & \text{otherwise}\ \end{cases}$
lt *- (17 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow \begin{cases} 1, & \text{if}\ a < b \ 0, & \text{otherwise}\ \end{cases}$
lte *- (18 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow \begin{cases} 1, & \text{if}\ a \le b \ 0, & \text{otherwise}\ \end{cases}$
gt *- (18 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow \begin{cases} 1, & \text{if}\ a > b \ 0, & \text{otherwise}\ \end{cases}$
gte *- (19 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow \begin{cases} 1, & \text{if}\ a \ge b \ 0, & \text{otherwise}\ \end{cases}$
is_odd *- (5 cycles)*	[a, ...]	[b, ...]	$b \leftarrow \begin{cases} 1, & \text{if}\ a \text{ is odd} \ 0, & \text{otherwise}\ \end{cases}$
eqw *- (15 cycles)*	[A, B, ...]	[c, A, B, ...]	$c \leftarrow \begin{cases} 1, & \text{if}\ a_i = b_i ; \forall i \in {0, 1, 2, 3} \ 0, & \text{otherwise}\ \end{cases}$

Extension Field Operations

Instruction	Stack Input	Stack Output	Notes
ext2add *- (5 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow (a + b) \mod q$
ext2sub *- (7 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow (a - b) \mod q$
ext2mul *- (3 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow (a \cdot b) \mod q$

ext2div *- (11 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow (a \cdot b^{-1}) \mod q$ Fails if $b = 0$
ext2neg *- (4 cycles)*	[a, ...]	[b, ...]	$b \leftarrow -a \mod q$
ext2inv *- (8 cycles)*	[a, ...]	[b, ...]	$b \leftarrow a^{-1} \mod q$ Fails if $a = 0$

where q is an irreducible polynomial $x^2 - x + 2$ over F_p for $p = 2^{64} - 2^{32} + 1$

id: flow_control title: Flow Control sidebar_label: Flow Control description: "Loops and conditionals." keywords: - docs - matic - polygon - miden - loop - conditional image: https://wiki.polygon.technology/img/thumbnail/polygon-miden.png

As mentioned above, Miden assembly provides high-level constructs to facilitate flow control. These constructs are:

- *if-else* expressions for conditional execution.
- *repeat* expressions for bounded counter-controlled loops.
- *while* expressions for unbounded condition-controlled loops.

Conditional execution

Conditional execution in Miden VM can be accomplished with *if-else* statements. These statements look like so:

```
if.true
    <instructions>
else
    <instructions>
end
```

where `instructions` can be a sequence of any instructions, including nested control structures; the `else` clause is optional. The above does the following:

1. Pops the top item from the stack.
2. If the value of the item is 1, instructions in the `if.true` branch are executed.
3. If the value of the item is 0, instructions in the `else` branch are executed.
4. If the value is not binary, the execution fails.

A note on performance: using *if-else* statements incurs a small, but non-negligible overhead. Thus, for simple conditional statements, it may be more efficient to compute the result of both branches, and then select the result using conditional drop instructions.

Counter-controlled loops

Executing a sequence of instructions a predefined number of times can be accomplished with *repeat* statements. These statements look like so:

```
repeat.<count>
    <instructions>
end
```

where:

- `instructions` can be a sequence of any instructions, including nested control structures.
- `count` is the number of times the `instructions` sequence should be repeated (e.g. `repeat.10`). `count` must be an integer greater than 0.

> ***Note**: During compilation the `repeat.<count>` blocks are unrolled and expanded into `<count>` copies of its inner block, there is no additional cost for counting variables in this case.*

Condition-controlled loops

Executing a sequence of instructions zero or more times based on some condition can be accomplished with *while loop* expressions. These expressions look like so:

```
while.true
    <instructions>
end
```

where `instructions` can be a sequence of any instructions, including nested control structures. The above does the following:

1. Pops the top item from the stack.
2. If the value of the item is 1, `instructions` in the loop body are executed. a. After the body is executed, the stack is popped again, and if the popped value is 1, the body is executed again. b. If the popped value is 0, the loop is exited. c. If the popped value is not binary, the execution fails.
3. If the value of the item is 0, execution of loop body is skipped.
4. If the value is not binary, the execution fails.

Example:

```
# push the boolean true to the stack
push.1

# pop the top element of the stack and loop while it is true
while.true
    # push the boolean false to the stack, finishing the loop for the next iteration
    push.0
end
```

---

id: io_operations title: Input / Output Operations sidebar_label: I/O Operations description: "Instructions for moving data between the stack and other sources." keywords: - docs - matic - polygon - miden - input - output image: https://wiki.polygon.technology/img/thumbnail/polygon-miden.png

Miden assembly provides a set of instructions for moving data between the stack and several other sources. These sources include:

- **Program code**: values to be moved onto the stack can be hard-coded in a program's source code.
- **Environment**: values can be moved onto the stack from environment variables. Currently, the available environment variables are *stack_depth*, which holds the current depth of the stack, and *local_address*, which stores absolute addresses of local variables. In the future, other environment variables may be added.
- **Advice stack**: values can be moved onto the stack from the advice provider by popping them from the advice stack. There is no limit on the number of values in the advice stack.
- **Memory**: values can be moved between the stack and random-access memory. The memory is word-addressable, meaning, four elements are located at each address, and we can read and write elements to/from memory in batches of four. Memory can be accessed via absolute memory references (i.e., via memory addresses) as well as via local procedure references (i.e., local index). The latter approach ensures that a procedure does not access locals of another procedure.

In the future several other sources such as *storage* and *logs* may be added.

Constant inputs

Instruction	Stack_input	Stack_output	Notes
push.*a* - *(1-2 cycles)* push.*a.b* push.*a.b.c*...	[...]	[a, ...] [b, a, ...] [c, b, a, ...]	Pushes values a, b, c etc. onto the stack. Up to 16 values can be specified. All values must be valid field elements in decimal (e.g., 123) or hexadecimal (e.g., $0x7b$) representation.

When specifying values in hexadecimal format, it is possible to omit the periods between individual values as long as total number of specified bytes is a multiple of 8. That is, the following are semantically equivalent:

```
push.0x1234.0xabcd
push.0x00000000000012340000000000abcd
```

In both case the values must still encode valid field elements.

Environment inputs

Instruction	Stack_input	Stack_output	Notes
sdepth - *(1 cycle)*	[...]	[d, ...]	$d \leftarrow stack.depth()$ Pushes the current depth of the stack onto the stack.
caller - *(1 cycle)*	[A, b, ...]	[H, b, ...]	$H \leftarrow context.fn_hash()$ Overwrites the top four stack items with the hash of a function which initiated the current SYSCALL. Executing this instruction outside of SYSCALL context will fail.
locaddr.*i* - *(2 cycles)*	[...]	[a, ...]	$a \leftarrow address_of(i)$ Pushes the absolute memory address of local memory at index i onto the stack.
clk - *(1 cycle)*	[...]	[t, ...]	$t \leftarrow clock_value()$ Pushes the current value of the clock cycle counter onto the stack.

Non-deterministic inputs

Instruction	Stack_input	Stack_output	Notes

adv_push.*n* *- (n cycles)*	[...]	[a, ...]	$a \leftarrow stack.pop()$ Pops n values from the advice stack and pushes them onto the operand stack. Valid for $n \in {1, ..., 16}$. Fails if the advice stack has fewer than n values.
adv_loadw *- (1 cycle)*	[0, 0, 0, 0, ...]	[A, ...]	$A \leftarrow stack.pop(4)$ Pop the next word (4 elements) from the advice stack and overwrites the first word of the operand stack (4 elements) with them. Fails if the advice stack has fewer than 4 values.
adv_pipe *- (2 cycles)*	[S2, S1, S0, a, ...]	[T2, T1, T0, b, ...]	$[T_0, T_1, T_2] \leftarrow permute(S_0, stack.pop(4), stack.pop(4))$ $b \leftarrow a + 2$ Pops the next two words (8 elements) from the advice stack, inserts them into memory at address a sequentially, overwrites these top 8 elements onto the operand stack, and performs a Rescue Prime Optimized permutation to the top 12 elements of the operand stack. At the end of the operation, the address is incremented by 2. Fails if the advice stack has fewer than 8 values.

> ***Note***: *The opcodes above always push data onto the operand stack so that the first element is placed deepest in the stack. For example, if the data on the stack is* `a,b,c,d` *and you use the opcode* `adv_push.4`*, the data will be* `d,c,b,a` *on your stack. This is also the behavior of the other opcodes.*

Random access memory

As mentioned above, there are two ways to access memory in Miden VM. The first way is via memory addresses using the instructions listed below. The addresses are absolute - i.e., they don't depend on the procedure context. Memory addresses can be in the range $[0, 2^{32})$.

Memory is guaranteed to be initialized to zeros. Thus, when reading from memory address which hasn't been written to previously, zero elements will be returned.

Instruction	Stack_input	Stack_output	Notes
mem_load *- (1 cycle)* mem_load.*a* *- (2 cycles)*	[a, ...]	[v, ...]	$v \leftarrow mem[a][0]$ Reads a word (4 elements) from memory at address *a*, and pushes the first element of the word onto the stack. If a is provided via the stack, it is removed from the stack first. Fails if $a \ge 2^{32}$
mem_loadw *- (1 cycle)* mem_loadw.*a* *- (2 cycles)*	[a, 0, 0, 0, 0, ...]	[A, ...]	$A \leftarrow mem[a]$ Reads a word from memory at address a and overwrites top four stack elements with it. If a is provided via the stack, it is removed from the stack first. Fails if $a \ge 2^{32}$
mem_store *- (2 cycles)* mem_store.*a* *- (3-4 cycles)*	[a, v, ...]	[...]	$v \rightarrow mem[a][0]$ Pops the top element off the stack and stores it as the first element of the word in memory at address a. All other elements of the word are not affected. If a is provided via the stack, it is removed from the stack first. Fails if $a \ge 2^{32}$
mem_storew *- (1 cycle)* mem_storew.*a* *- (2-3 cycles)*	[a, A, ...]	[A, ...]	$A \rightarrow mem[a]$ Stores the top four elements of the stack in memory at address a. If a is provided via the stack, it is removed from the stack first. Fails if $a \ge 2^{32}$
mem_stream *- (2 cycles)*	[S2, S1, S0, a, ...]	[T2, T1, T0, b, ...]	$[T_0, T_1, T_2] \leftarrow permute(S_0, mem[a], mem[a+1])$ $b \leftarrow a + 2$ Loads two words from memory starting at the address a, overwrites the top 8 elements of the stack with them, and applies Rescue Prime Optimized permutation to the top 12 elements of the stack. At the end of the operation the address is incremented by 2.

The second way to access memory is via procedure locals using the instructions listed below. These instructions are available only in procedure context. The number of locals available to a given procedure must be specified at procedure declaration time, and trying to access more locals than was declared will result in a

compile-time error. The number of locals per procedure is not limited, but the total number of locals available to all procedures at runtime must be smaller than 2^{32}.

Instruction	Stack_input	Stack_output	Notes
loc_load.*i* *- (3-4 cycles)*	[...]	[v, ...]	$v \leftarrow local[i][0]$ Reads a word (4 elements) from local memory at index *i*, and pushes the first element of the word onto the stack.
loc_loadw.*i* *- (3-4 cycles)*	[0, 0, 0, 0, ...]	[A, ...]	$A \leftarrow local[i]$ Reads a word from local memory at index i and overwrites top four stack elements with it.
loc_store.*i* *- (4-5 cycles)*	[v, ...]	[...]	$v \rightarrow local[i][0]$ Pops the top element off the stack and stores it as the first element of the word in local memory at index i. All other elements of the word are not affected.
loc_storew.*i* *- (3-4 cycles)*	[A, ...]	[A, ...]	$A \rightarrow local[i]$ Stores the top four elements of the stack in local memory at index i.

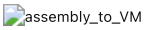
Unlike regular memory, procedure locals are not guaranteed to be initialized to zeros. Thus, when working with locals, one must assume that before a local memory address has been written to, it contains "garbage".

Internally in the VM, procedure locals are stored at memory offset stating at 2^{30}. Thus, every procedure local has an absolute address in regular memory. The `locaddr.i` instruction is provided specifically to map an index of a procedure's local to an absolute address so that it can be passed to downstream procedures, when needed.

id: main title: Miden Assembly Language sidebar_label: Overview description: "The Miden language's advantages, notations and design." keywords: - docs - matic - polygon - miden - assembly image: https://wiki.polygon.technology/img/thumbnail/polygon-miden.png

Miden assembly is a simple, low-level language for writing programs for Miden VM. It stands just above raw Miden VM instruction set, and in fact, many instructions of Miden assembly map directly to raw instructions of Miden VM.

Before Miden assembly can be executed on Miden VM, it needs to be compiled into a Program MAST (Merkelized Abstract Syntax Tree) which is a binary tree of code blocks each containing raw Miden VM instructions.


assembly_to_VM

As compared to raw Miden VM instructions, Miden assembly has several advantages:

1. Miden assembly is intended to be a more stable external interface for the VM. That is, while we plan to make significant changes to the underlying VM to optimize it for stability, performance etc., we intend to make very few breaking changes to Miden assembly.
2. Miden assembly natively supports control flow expressions which the assembler automatically transforms into a program MAST. This greatly simplifies writing programs with complex execution logic.
3. Miden assembly supports *macro instructions*. These instructions expand into short sequences of raw Miden VM instructions making it easier to encode common operations.
4. Miden assembly supports *procedures*. These are stand-alone blocks of code which the assembler inlines into program MAST at compile time. This improves program modularity and code organization.

The last two points also make Miden assembly much more concise as compared to the raw program MAST. This may be important in the blockchain context where pubic programs need to be stored on chain.

Terms and notations

In this document we use the following terms and notations:

- p is the modulus of the VM's base field which is equal to $2^{64} - 2^{32} + 1$.
- A *binary* value means a field element which is either 0 or 1.
- Inequality comparisons are assumed to be performed on integer representations of field elements in the range $[0, p)$.

Throughout this document, we use lower-case letters to refer to individual field elements (e.g., a). Sometimes it is convenient to describe operations over groups of elements. For these purposes we define a *word* to be a group of four elements. We use upper-case letters to refer to words (e.g., A). To refer to individual elements within a word, we use numerical subscripts. For example, a_0 is the first element of word A, b_3 is the last element of word B, etc.

Design goals

The design of Miden assembly tries to achieve the following goals:

1. Miden assembly should be an easy compilation target for high-level languages.
2. Programs written in Miden assembly should be readable, even if the code is generated by a compiler from a high-level language.

3. Control flow should be easy to understand to help in manual inspection, formal verification, and optimization.
4. Compilation of Miden assembly into Miden program MAST should be as straight-forward as possible.
5. Serialization of Miden assembly into a binary representation should be as compact and as straight-forward as possible.

In order to achieve the first goal, Miden assembly exposes a set of native operations over 32-bit integers and supports linear read-write memory. Thus, from the stand-point of a higher-level language compiler, Miden VM can be viewed as a regular 32-bit stack machine with linear read-write memory.

In order to achieve the second and third goals, Miden assembly facilitates flow control via high-level constructs like `while` loops, `if-else` statements, and function calls with statically defined targets. Thus, for example, there are no explicit `jump` instructions.

In order to achieve the fourth goal, Miden assembly retains direct access to the VM stack rather than abstracting it away with higher-level constructs and named variables.

Lastly, in order to achieve the fifth goal, each instruction of Miden assembly can be encoded using a single byte. The resulting byte-code is simply a one-to-one mapping of instructions to their binary values.

id: stack_manipulation title: Stack Manipulation sidebar_label: Stack Manipulation description: "Instructions for stack manipulation." keywords: - docs - matic - polygon - miden - stack - manipulation image: https://wiki.polygon.technology/img/thumbnail/polygon-miden.png

Miden VM stack is a push-down stack of field elements. The stack has a maximum depth of 2^{32}, but only the top 16 elements are directly accessible via the instructions listed below.

In addition to the typical stack manipulation instructions such as `drop`, `dup`, `swap` etc., Miden assembly provides several conditional instructions which can be used to manipulate the stack based on some condition - e.g., conditional swap `cswap` or conditional drop `cdrop`.

Instruction	Stack_input	Stack_output	Notes
drop *- (1 cycle)*	[a, ...]	[...]	Deletes the top stack item.
dropw *- (4 cycles)*	[A, ...]	[...]	Deletes a word (4 elements) from the top of the stack.
padw *- (4 cycles)*	[...]	[0, 0, 0, 0, ...]	Pushes four 0 values onto the stack. Note: simple `pad` is not provided because `push.0` does the same thing.
dup.*n* *- (1-3 cycles)*	[..., a, ...]	[a, ..., a, ...]	Pushes a copy of the nth stack item onto the stack. `dup` and `dup.0` are the same instruction. Valid for $n \in {0, ..., 15}$
dupw.*n* *- (4 cycles)*	[..., A, ...]	[A, ..., A, ...]	Pushes a copy of the nth stack word onto the stack. `dupw` and `dupw.0` are the same instruction. Valid for $n \in {0, 1, 2, 3}$
swap.*n* *- (1-6 cycles)*	[a, ..., b, ...]	[b, ..., a, ...]	Swaps the top stack item with the nth stack item. `swap` and `swap.1` are the same instruction. Valid for $n \in {1, ..., 15}$
swapw.*n* *- (1 cycle)*	[A, ..., B, ...]	[B, ..., A, ...]	Swaps the top stack word with the nth stack word. `swapw` and `swapw.1` are the same instruction. Valid for $n \in {1, 2, 3}$
movup.*n* *- (1-4 cycles)*	[..., a, ...]	[a, ...]	Moves the nth stack item to the top of the stack. Valid for $n \in {2, ..., 15}$
movupw.*n* *- (2-3 cycles)*	[..., A, ...]	[A, ...]	Moves the nth stack word to the top of the stack. Valid for $n \in {2, 3}$
movdn.*n* *- (1-4 cycles)*	[a, ...]	[..., a, ...]	Moves the top stack item to the nth position of the stack. Valid for $n \in {2, ..., 15}$
movdnw.*n* *- (2-3 cycles)*	[A, ...]	[..., A, ...]	Moves the top stack word to the nth word position of the stack. Valid for $n \in {2, 3}$

Conditional manipulation

Instruction	Stack_input	Stack_output	Notes
cswap *- (1 cycle)*	[c, b, a, ...]	[e, d, ...]	$d = \begin{cases} a, & \text{if}\ c = 0 \ b, & \text{if}\ c = 1\ \end{cases}$ $e = \begin{cases} b, & \text{if}\ c = 0 \ a, & \text{if}\ c = 1\ \end{cases}$ Fails if $c > 1$
cswapw *- (1 cycle)*	[c, B, A, ...]	[E, D, ...]	$D = \begin{cases} A, & \text{if}\ c = 0 \ B, & \text{if}\ c = 1\ \end{cases}$ $E = \begin{cases} B, & \text{if}\ c = 0 \ A, & \text{if}\ c = 1\ \end{cases}$ Fails if $c > 1$
cdrop *- (2 cycles)*	[c, b, a, ...]	[d, ...]	$d = \begin{cases} a, & \text{if}\ c = 0 \ b, & \text{if}\ c = 1\ \end{cases}$ Fails if $c > 1$
cdropw *- (5 cycles)*	[c, B, A, ...]	[D, ...]	$D = \begin{cases} A, & \text{if}\ c = 0 \ B, & \text{if}\ c = 1\ \end{cases}$ Fails if $c > 1$

---

id: u32_operations title: u32 Operations sidebar_label: u32 Operations description: "Instructions for operations on 32-bit integers." keywords: - docs - matic - polygon - miden - u32 image: https://wiki.polygon.technology/img/thumbnail/polygon-miden.png

Miden assembly provides a set of instructions which can perform operations on regular two-complement 32-bit integers. These instructions are described in the tables below.

Most instructions have *checked* variants. These variants ensure that input values are 32-bit integers, and fail if that's not the case. All other variants do not perform these checks, and thus, should be used only if the inputs are known to be 32-bit integers. Supplying inputs which are greater than or equal to 2^{32} to unchecked operations results in undefined behavior.

The primary benefit of using unchecked operations is performance: they can frequently be executed 2 or 3 times faster than their checked counterparts. In general, vast majority of the unchecked operations listed below can be executed in a single VM cycle.

For instructions where one or more operands can be provided as immediate parameters (e.g., `u32checked_add` and `u32checked_add.b`), we provide stack transition diagrams only for the non-immediate version. For the immediate version, it can be assumed that the operand with the specified name is not present on the stack.

In all the table below, the number of cycles it takes for the VM to execute each instruction is listed beneath the instruction.

Conversions and tests

Instruction	Stack input	Stack output	Notes
u32test *- (5 cycles)*	[a, ...]	[b, a, ...]	$b \leftarrow \begin{cases} 1, & \text{if}\ a < 2^{32} \ 0, & \text{otherwise}\ \end{cases}$
u32testw *- (23 cycles)*	[A, ...]	[b, A, ...]	$b \leftarrow \begin{cases} 1, & \text{if}\ \forall\ i \in {0, 1, 2, 3}\ a_i < 2^{32} \ 0, & \text{otherwise}\ \end{cases}$
u32assert u32assert.1 *- (3 cycles)*	[a, ...]	[a, ...]	Fails if $a \ge 2^{32}$
u32assert.2 *- (1 cycle)*	[b, a,...]	[b, a,...]	Fails if $a \ge 2^{32}$ or $b \ge 2^{32}$
u32assertw *- (6 cycles)*	[A, ...]	[A, ...]	Fails if $\exists\ i \in {0, 1, 2, 3} \ni a_i \ge 2^{32}$
u32cast *- (2 cycles)*	[a, ...]	[b, ...]	$b \leftarrow a \mod 2^{32}$
u32split *- (1 cycle)*	[a, ...]	[c, b, ...]	$b \leftarrow a \mod 2^{32}$, $c \leftarrow \lfloor{a / 2^{32}}\rfloor$

Arithmetic operations

Instruction	Stack input	Stack output	Notes
u32checked_add - *(4 cycles)* u32checked_add.*b* - *(5-6 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow a + b$ Fails if $max(a, b, c) \ge 2^{32}$
u32overflowing_add - *(1 cycle)* u32overflowing_add.*b* - *(2-3 cycles)*	[b, a, ...]	[d, c, ...]	$c \leftarrow (a + b) \mod 2^{32}$ $d \leftarrow \begin{cases} 1, & \text{if}\ (a + b) \ge 2^{32} \ 0, & \text{otherwise}\ \end{cases}$ Undefined if $max(a, b) \ge 2^{32}$
u32wrapping_add - *(2 cycles)* u32wrapping_add.*b* - *(3-4 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow (a + b) \mod 2^{32}$ Undefined if $max(a, b) \ge 2^{32}$
u32overflowing_add3 - *(1 cycle)*	[c, b, a, ...]	[e, d, ...]	$d \leftarrow (a + b + c) \mod 2^{32}$, $e \leftarrow \lfloor (a + b + c) / 2^{32}\rfloor$ Undefined if $max(a, b, c) \ge 2^{32}$
u32wrapping_add3 - *(2 cycles)*	[c, b, a, ...]	[d, ...]	$d \leftarrow (a + b + c) \mod 2^{32}$, Undefined if $max(a, b, c) \ge 2^{32}$
u32checked_sub - *(4 cycles)* u32checked_sub.*b* - *(5-6 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow (a - b)$ Fails if $max(a, b) \ge 2^{32}$ or $a < b$
u32overflowing_sub - *(1 cycle)* u32overflowing_sub.*b* - *(2-3 cycles)*	[b, a, ...]	[d, c, ...]	$c \leftarrow (a - b) \mod 2^{32}$ $d \leftarrow \begin{cases} 1, & \text{if}\ a < b \ 0, & \text{otherwise}\ \end{cases}$ Undefined if $max(a, b) \ge 2^{32}$
u32wrapping_sub - *(2 cycles)* u32wrapping_sub.*b* - *(3-4 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow (a - b) \mod 2^{32}$ Undefined if $max(a, b) \ge 2^{32}$
u32checked_mul - *(4 cycles)* u32checked_mul.*b* - *(5-6 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow a \cdot b$ Fails if $max(a, b, c) \ge 2^{32}$
u32overflowing_mul - *(1 cycle)* u32overflowing_mul.*b*	[b, a, ...]	[d, c, ...]	$c \leftarrow (a \cdot b) \mod 2^{32}$ $d \leftarrow \lfloor(a \cdot b) / 2^{32}\rfloor$ Undefined if $max(a, b) \ge 2^{32}$

- (2-3 cycles)			
u32wrapping_mul *- (2 cycles)* u32wrapping_mul.*b* *- (3-4 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow (a \cdot b) \mod 2^{32}$ Undefined if $max(a, b) \ge 2^{32}$
u32overflowing_madd *- (1 cycle)*	[b, a, c, ...]	[e, d, ...]	$d \leftarrow (a \cdot b + c) \mod 2^{32}$ $e \leftarrow \lfloor(a \cdot b + c) / 2^{32}\rfloor$ Undefined if $max(a, b, c) \ge 2^{32}$
u32wrapping_madd *- (2 cycles)*	[b, a, c, ...]	[d, ...]	$d \leftarrow (a \cdot b + c) \mod 2^{32}$ Undefined if $max(a, b, c) \ge 2^{32}$
u32checked_div *- (3 cycles)* u32checked_div.*b* *- (4-5 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow \lfloor a / b\rfloor$ Fails if $max(a, b) \ge 2^{32}$ or $b = 0$
u32unchecked_div *- (2 cycles)* u32unchecked_div.*b* *- (3-4 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow \lfloor a / b\rfloor$ Fails if $b = 0$ Undefined if $max(a, b) \ge 2^{32}$
u32checked_mod *- (4 cycles)* u32checked_mod.*b* *- (5-6 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow a \mod b$ Fails if $max(a, b) \ge 2^{32}$ or $b = 0$
u32unchecked_mod *- (3 cycles)* u32unchecked_mod.*b* *- (4-5 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow a \mod b$ Fails if $b = 0$ Undefined if $max(a, b) \ge 2^{32}$
u32checked_divmod *- (2 cycles)* u32checked_divmod.*b* *- (3-4 cycles)*	[b, a, ...]	[d, c, ...]	$c \leftarrow \lfloor a / b\rfloor$ $d \leftarrow a \mod b$ Fails if $max(a, b) \ge 2^{32}$ or $b = 0$
u32unchecked_divmod *- (1 cycle)* u32unchecked_divmod.*b* *- (2-3 cycles)*	[b, a, ...]	[d, c, ...]	$c \leftarrow \lfloor a / b\rfloor$ $d \leftarrow a \mod b$ Fails if $b = 0$ Undefined if $max(a, b) \ge 2^{32}$

Bitwise operations

Instruction	Stack input	Stack output	Notes
u32checked_and *- (1 cycle)*	[b, a, ...]	[c, ...]	Computes c as a bitwise AND of binary representations of a and b. Fails if $max(a,b) \ge 2^{32}$
u32checked_or	[b, a, ...]	[c, ...]	Computes c as a bitwise OR of binary representations of a and b.

- (6 cycle)s			Fails if $max(a,b) \ge 2^{32}$
u32checked_xor *- (1 cycle)*	[b, a, ...]	[c, ...]	Computes c as a bitwise XOR of binary representations of a and b. Fails if $max(a,b) \ge 2^{32}$
u32checked_not *- (5 cycles)*	[a, ...]	[b, ...]	Computes b as a bitwise NOT of binary representation of a. Fails if $a \ge 2^{32}$
u32checked_shl *- (47 cycles)* u32checked_shl.*b* *- (4 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow (a \cdot 2^b) \mod 2^{32}$ Fails if $a \ge 2^{32}$ or $b > 31$
u32unchecked_shl *- (40 cycles)* u32unchecked_shl.*b* *- (3 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow (a \cdot 2^b) \mod 2^{32}$ Undefined if $a \ge 2^{32}$ or $b > 31$
u32checked_shr *- (47 cycles)* u32checked_shr.*b* *- (4 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow \lfloor a/2^b \rfloor$ Fails if $a \ge 2^{32}$ or $b > 31$
u32unchecked_shr *- (40 cycles)* u32unchecked_shr.*b* *- (3 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow \lfloor a/2^b \rfloor$ Undefined if $a \ge 2^{32}$ or $b > 31$
u32checked_rotl *- (47 cycles)* u32checked_rotl.*b* *- (4 cycles)*	[b, a, ...]	[c, ...]	Computes c by rotating a 32-bit representation of a to the left by b bits. Fails if $a \ge 2^{32}$ or $b > 31$
u32unchecked_rotl *- (40 cycles)* u32unchecked_rotl.*b* *- (3 cycles)*	[b, a, ...]	[c, ...]	Computes c by rotating a 32-bit representation of a to the left by b bits. Undefined if $a \ge 2^{32}$ or $b > 31$
u32checked_rotr *- (59 cycles)* u32checked_rotr.*b* *- (6 cycles)*	[b, a, ...]	[c, ...]	Computes c by rotating a 32-bit representation of a to the right by b bits. Fails if $a \ge 2^{32}$ or $b > 31$
u32unchecked_rotr *- (44 cycles)* u32unchecked_rotr.*b* *- (3 cycles)*	[b, a, ...]	[c, ...]	Computes c by rotating a 32-bit representation of a to the right by b bits. Undefined if $a \ge 2^{32}$ or $b > 31$
u32checked_popcnt *- (36 cycles)*	[a, ...]	[b, ...]	Computes b by counting the number of set bits in a (hamming weight of a). Fails if $a \ge 2^{32}$
u32unchecked_popcnt	[a, ...]	[b, ...]	Computes b by counting the number of set bits in a (hamming weight of a).

- (33 cycles)			Undefined if $a \ge 2^{32}$

Comparison operations

Instruction	Stack input	Stack output	Notes
u32checked_eq *- (2 cycles)* u32checked_eq.*b* *- (3-4 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow \begin{cases} 1, & \text{if}\ a=b \ 0, & \text{otherwise}\ \end{cases}$ Fails if $max(a, b) \ge 2^{32}$ Note: unchecked version is not provided because it is equivalent to simple `eq`.
u32checked_neq *- (3 cycles)* u32checked_neq.*b* *- (4-5 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow \begin{cases} 1, & \text{if}\ a \ne b \ 0, & \text{otherwise}\ \end{cases}$ Fails if $max(a, b) \ge 2^{32}$ Note: unchecked version is not provided because it is equivalent to simple `neq`.
u32checked_lt *- (6 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow \begin{cases} 1, & \text{if}\ a < b \ 0, & \text{otherwise}\ \end{cases}$ Fails if $max(a, b) \ge 2^{32}$
u32unchecked_lt *- (5 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow \begin{cases} 1, & \text{if}\ a < b \ 0, & \text{otherwise}\ \end{cases}$ Undefined if $max(a, b) \ge 2^{32}$
u32checked_lte *- (8 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow \begin{cases} 1, & \text{if}\ a \le b \ 0, & \text{otherwise}\ \end{cases}$ Fails if $max(a, b) \ge 2^{32}$
u32unchecked_lte *- (7 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow \begin{cases} 1, & \text{if}\ a \le b \ 0, & \text{otherwise}\ \end{cases}$ Undefined if $max(a, b) \ge 2^{32}$
u32checked_gt *- (7 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow \begin{cases} 1, & \text{if}\ a > b \ 0, & \text{otherwise}\ \end{cases}$ Fails if $max(a, b) \ge 2^{32}$
u32unchecked_gt *- (6 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow \begin{cases} 1, & \text{if}\ a > b \ 0, & \text{otherwise}\ \end{cases}$ Undefined if $max(a, b) \ge 2^{32}$
u32checked_gte *- (7 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow \begin{cases} 1, & \text{if}\ a \ge b \ 0, & \text{otherwise}\ \end{cases}$ Fails if $max(a, b) \ge 2^{32}$
u32unchecked_gte *- (6 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow \begin{cases} 1, & \text{if}\ a \ge b \ 0, & \text{otherwise}\ \end{cases}$ Undefined if $max(a, b) \ge 2^{32}$
u32checked_min *- (9 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow \begin{cases} a, & \text{if}\ a < b \ b, & \text{otherwise}\ \end{cases}$ Fails if $max(a, b) \ge 2^{32}$
u32unchecked_min *- (8 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow \begin{cases} a, & \text{if}\ a < b \ b, & \text{otherwise}\ \end{cases}$ Undefined if $max(a, b) \ge 2^{32}$
u32checked_max *- (10 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow \begin{cases} a, & \text{if}\ a > b \ b, & \text{otherwise}\ \end{cases}$ Fails if $max(a, b) \ge 2^{32}$
u32unchecked_max *- (9 cycles)*	[b, a, ...]	[c, ...]	$c \leftarrow \begin{cases} a, & \text{if}\ a > b \ b, & \text{otherwise}\ \end{cases}$ Undefined if $max(a, b) \ge 2^{32}$

---

id: fri title: FRI Verification Procedures sidebar_label: FRI Verification description: FRI Module contains procedures for verifying FRI proofs. keywords: - docs - polygon - wiki - miden - fri image: https://wiki.polygon.technology/img/thumbnail/polygon-miden.png

Namespace `std::crypto::fri` contains modules for verifying FRI proofs.

FRI Extension 2, Fold 4

Module `std::crypto::fri::frie2f4` contains procedures for verifying FRI proofs generated over the quadratic extension of the Miden VM's base field. Moreover, the procedures assume that layer folding during the commit phase of FRI protocol was performed using folding factor 4.

Procedure	Description
verify	Verifies a FRI proof where the proof was generated over the quadratic extension of the base field and layer folding was performed using folding factor 4. Input: `[query_start_ptr, query_end_ptr, layer_ptr, rem_ptr, g, ...]`>> Output: `[...]` - `query_start_ptr` is a pointer to a list of tuples of the form `(e0, e1, p, 0)` where `p` is a query index at the first layer and `(e0, e1)` is an extension field element corresponding to the value of the first layer at index p. - `query_end_ptr` is a pointer to the first empty memory address after the last `(e0, e1, p, 0)` tuple. - `layer_ptr` is a pointer to the first layer commitment denoted throughout the code by C. `layer_ptr + 1` points to the first `(alpha0, alpha1, t_depth, d_size)` where `d_size` is the size of initial domain divided by 4, `t_depth` is the depth of the Merkle tree commitment to the first layer and `(alpha0, alpha1)` is the first challenge used in folding the first layer. Both `t_depth` and `d_size` are expected to be smaller than 2^32. Otherwise, the result of this procedure is undefined. - `rem_ptr` is a pointer to the first tuple of two consecutive degree 2 extension field elements making up the remainder codeword. This codeword can be of length either 32 or 64. The memory referenced above is used contiguously, as follows: `[layer_ptr ... rem_ptr ... query_start_ptr ... query_end_ptr]` This means for example that: 1. `rem_ptr - 1` points to the last `(alpha0, alpha1, t_depth, d_size)` tuple. 2. The length of the remainder codeword is `2 * (rem_ptr - query_start_ptr)`. Cycles: for domains of size `2^n` where: - `n` is even: 12 + 6 + num_queries * (40 + num_layers * 76 + 69) + 2626 - `n` is odd: 12 + 6 + num_queries * (40 + num_layers * 76 + 69) + 1356

---

id: hashes title: Cryptographic Hashes sidebar_label: Cryptographic Hashes description: Crypto module contains modules for commonly used cryptographic hash functions. keywords: - docs - polygon wiki - miden - hash - BLAKE3 - SHA256 - crypto image: https://wiki.polygon.technology/img/thumbnail/polygon-miden.png

Namespace `std::crypto` contains modules for commonly used cryptographic hash functions.

BLAKE3

Module `std::crypto::hashes::blake3` contains procedures for computing hashes using BLAKE3 hash function. The input and output elements are assumed to contain one 32-bit value per element.

Procedure	Description
hash	Computes BLAKE3 2-to-1 hash. Input: 64-bytes stored in the first 16 elements of the stack (32 bits per element). Output: A 32-byte digest stored in the first 8 elements of stack (32 bits per element) with the remaining 8 elements of stack set to zeros.

SHA256

Module `std::crypto::hashes::sha256` contains procedures for computing hashes using SHA256 hash function. The input and output elements are assumed to contain one 32-bit value per element.

Procedure	Description
hash	Computes SHA256 2-to-1 hash. Input: 64-bytes stored in the first 16 elements of the stack (32 bits per element). Output: A 32-byte digest stored in the first 8 elements of stack (32 bits per element) with the remaining 8 elements of stack set to zeros.

---

id: main title: Miden Standard Library sidebar_label: Overview description: "Organization, usage and available modules." keywords: - docs - matic - polygon - miden - library image: https://wiki.polygon.technology/img/thumbnail/polygon-miden.png

Miden standard library provides a set of procedures which can be used by any Miden program. These procedures build on the core instruction set of Miden assembly expanding the functionality immediately available to the user.

The goals of Miden standard library are:

- Provide highly-optimized and battle-tested implementations of commonly-used primitives.
- Reduce the amount of code that needs to be shared between parties for proving and verifying program execution.

The second goal can be achieved because calls to procedures in the standard library can always be serialized as 32 bytes, regardless of how large the procedure is.

Organization and usage

Procedures in the Miden Standard Library are organized into modules, each targeting a narrow set of functionality. Modules are grouped into higher-level namespaces. However, higher-level namespaces do not expose any procedures themselves. For example, `std::math::u64` is a module containing procedures for working with 64-bit unsigned integers. This module is a part of the `std::math` namespace. However, the `std::math` namespace does not expose any procedures.

For an example of how to invoke procedures from imported modules see this section.

Available modules

Currently, Miden standard library contains just a few modules, which are listed below. Over time, we plan to add many more modules which will include various cryptographic primitives, additional numeric data types and operations, and many others.

Module	Description
std::crypto::fri::frie2f4	Contains procedures for verifying FRI proofs (field extension = 2, folding factor = 4).
std::crypto::hashes::blake3	Contains procedures for computing hashes using BLAKE3 hash function.
std::crypto::hashes::sha256	Contains procedures for computing hashes using SHA256 hash function.
std::math::u64	Contains procedures for working with 64-bit unsigned integers.
std::mem	Contains procedures for working with random access memory.
std::sys	Contains system-level utility procedures.

---

id: u64 title: Unsigned 64-bit Integer Operations sidebar_label: Integer Operations description: Math module contains procedures to perform unsigned 64-bit integer operations. keywords: - docs - matic - polygon - miden - Arithmetic - Comparison - Bitwise image: https://wiki.polygon.technology/img/thumbnail/polygon-miden.png

Module `std::math::u64` contains a set of procedures which can be used to perform unsigned 64-bit integer operations. These operations fall into the following categories:

- **Arithmetic operations** - addition, multiplication, division etc.
- **Comparison operations** - equality, less than, greater than etc.
- **Bitwise operations** - binary AND, OR, XOR, bit shifts etc.

All procedures assume that an unsigned 64-bit integer (u64) is encoded using two elements, each containing an unsigned 32-bit integer (u32). When placed on the stack, the least-significant limb is assumed to be deeper in the stack. For example, a u64 value `a` consisting of limbs `a_hi` and `a_lo` would be position on the stack like so:

```
[a_hi, a_lo, ... ]
```

Procedures which check whether the input values are encoded correctly are designated with `checked` prefix. For example, `checked_add` would fail if any of the top 4 elements on the stack contains a value greater than $2^{32} - 1$. In contrast, `wrapping_add` and `overflowing_add` would not perform these checks, and therefore, if any of the top 4 stack elements is greater than $2^{32} - 1$, the operation will not fail but rather will produce an undefined result. Thus, when using versions of procedures which are not checked, it is important to be certain that input values are 32-bit limbs encoding valid u64 values.

Arithmetic operations

Procedure	Description
checked_add	Performs addition of two unsigned 64-bit integers and fails if the result would overflow. The input values are expected to be represented using 32-bit limbs, and the procedure will fail if they are not. The stack transition looks as follows: [b_hi, b_lo, a_hi, a_lo, ...] -> [c_hi, c_lo, ...], where c = (a + b) % 2^64
overflowing_add	Performs addition of two unsigned 64-bit integers preserving the overflow. The input values are assumed to be represented using 32-bit limbs, but this is not checked. The stack transition looks as follows: [b_hi, b_lo, a_hi, a_lo, ...] -> [overflow_flag, c_hi, c_lo, ...], where c = (a + b) % 2^64
wrapping_add	Performs addition of two unsigned 64-bit integers discarding the overflow. The input values are assumed to be represented using 32-bit limbs, but this is not checked. The stack transition looks as follows: [b_hi, b_lo, a_hi, a_lo, ...] -> [c_hi, c_lo, ...], where c = (a + b) % 2^64

checked_sub	Performs subtraction of two unsigned 64-bit integers and fails if the result would underflow. The input values are expected to be represented using 32-bit limbs, and the procedure will fail if they are not. The stack transition looks as follows: [b_hi, b_lo, a_hi, a_lo, ...] -> [c_hi, c_lo, ...], where c = (a - b) % 2^64
overflowing_sub	Performs subtraction of two unsigned 64-bit integers preserving the overflow. The input values are assumed to be represented using 32-bit limbs, but this is not checked. The stack transition looks as follows: [b_hi, b_lo, a_hi, a_lo, ...] -> [underflow_flag, c_hi, c_lo, ...], where c = (a - b) % 2^64
wrapping_sub	Performs subtraction of two unsigned 64-bit integers discarding the overflow. The input values are assumed to be represented using 32-bit limbs, but this is not checked. The stack transition looks as follows: [b_hi, b_lo, a_hi, a_lo, ...] -> [c_hi, c_lo, ...], where c = (a - b) % 2^64
checked_mul	Performs multiplication of two unsigned 64-bit integers and fails if the result would overflow. The input values are expected to be represented using 32-bit limbs, and the procedure will fail if they are not. The stack transition looks as follows: [b_hi, b_lo, a_hi, a_lo, ...] -> [c_hi, c_lo, ...], where c = (a * b) % 2^64
overflowing_mul	Performs multiplication of two unsigned 64-bit integers preserving the overflow. The input values are assumed to be represented using 32-bit limbs, but this is not checked. The stack transition looks as follows: [b_hi, b_lo, a_hi, a_lo, ...] -> [c_hi_hi, c_hi_lo, c_lo_hi, c_lo_lo, ...], where c = (a * b) % 2^64
wrapping_mul	Performs multiplication of two unsigned 64-bit integers discarding the overflow. The input values are assumed to be represented using 32-bit limbs, but this is not checked. The stack transition looks as follows: [b_hi, b_lo, a_hi, a_lo, ...] -> [c_hi, c_lo, ...], where c = (a * b) % 2^64
checked_div	Performs division of two unsigned 64-bit integers discarding the remainder. The input values are expected to be represented using 32-bit limbs, and the procedure will fail if they are not. The stack transition looks as follows: [b_hi, b_lo, a_hi, a_lo, ...] -> [c_hi, c_lo, ...], where c = a // b
unchecked_div	Performs division of two unsigned 64-bit integers discarding the remainder. The input values are assumed to be represented using 32-bit limbs, but this is not checked. The stack transition looks as follows: [b_hi, b_lo, a_hi, a_lo, ...] -> [c_hi, c_lo, ...], where c = a // b
checked_mod	Performs modulo operation of two unsigned 64-bit integers. The input values are expected to be represented using 32-bit limbs, and the procedure will fail if they are not. The stack transition looks as follows: [b_hi, b_lo, a_hi, a_lo, ...] -> [c_hi, c_lo, ...], where c = a % b
unchecked_mod	Performs modulo operation of two unsigned 64-bit integers. The input values are assumed to be represented using 32-bit limbs, but this is not checked. The stack transition looks as follows: [b_hi, b_lo, a_hi, a_lo, ...] -> [c_hi, c_lo, ...], where c = a % b
checked_divmod	Performs divmod operation of two unsigned 64-bit integers. The input values are expected to be represented using 32-bit limbs, and the procedure will fail if they are not. The stack transition looks as follows: [b_hi, b_lo, a_hi, a_lo, ...] -> [r_hi, r_lo, q_hi, q_lo ...], where r = a % b, q = a // b
unchecked_divmod	Performs divmod operation of two unsigned 64-bit integers. The input values are assumed to be represented using 32-bit limbs, but this is not checked. The stack transition looks as follows: [b_hi, b_lo, a_hi, a_lo, ...] -> [r_hi, r_lo, q_hi, q_lo ...], where r = a % b, q = a // b

Comparison operations

Procedure	Description
checked_lt	Performs less-than comparison of two unsigned 64-bit integers. The input values are expected to be represented using 32-bit limbs, and the procedure will fail if they are not. The stack transition looks as follows: [b_hi, b_lo, a_hi, a_lo, ...] -> [c, ...], where c = 1 when a < b, and 0 otherwise.
unchecked_lt	Performs less-than comparison of two unsigned 64-bit integers. The input values are assumed to be represented using 32-bit limbs, but this is not checked. The stack transition looks as follows: [b_hi, b_lo, a_hi, a_lo, ...] -> [c, ...], where c = 1 when a < b, and 0 otherwise.
checked_gt	Performs greater-than comparison of two unsigned 64-bit integers. The input values are expected to be represented using 32-bit limbs, and the procedure will fail if they are not. The stack transition looks as follows: [b_hi, b_lo, a_hi, a_lo, ...] -> [c, ...], where c = 1 when a > b, and 0 otherwise.

unchecked_gt	Performs greater-than comparison of two unsigned 64-bit integers. The input values are assumed to be represented using 32-bit limbs, but this is not checked. The stack transition looks as follows: [b_hi, b_lo, a_hi, a_lo, ...] -> [c, ...], where c = 1 when a > b, and 0 otherwise. This takes 11 cycles.
checked_lte	Performs less-than-or-equal comparison of two unsigned 64-bit integers. The input values are expected to be represented using 32-bit limbs, and the procedure will fail if they are not. The stack transition looks as follows: [b_hi, b_lo, a_hi, a_lo, ...] -> [c, ...], where c = 1 when a <= b, and 0 otherwise.
unchecked_lte	Performs less-than-or-equal comparison of two unsigned 64-bit integers. The input values are assumed to be represented using 32-bit limbs, but this is not checked. The stack transition looks as follows: [b_hi, b_lo, a_hi, a_lo, ...] -> [c, ...], where c = 1 when a <= b, and 0 otherwise.
checked_gte	Performs greater-than-or-equal comparison of two unsigned 64-bit integers. The input values are expected to be represented using 32-bit limbs, and the procedure will fail if they are not. The stack transition looks as follows: [b_hi, b_lo, a_hi, a_lo, ...] -> [c, ...], where c = 1 when a >= b, and 0 otherwise.
unchecked_gte	Performs greater-than-or-equal comparison of two unsigned 64-bit integers. The input values are assumed to be represented using 32-bit limbs, but this is not checked. The stack transition looks as follows: [b_hi, b_lo, a_hi, a_lo, ...] -> [c, ...], where c = 1 when a >= b, and 0 otherwise.
checked_eq	Performs equality comparison of two unsigned 64-bit integers. The input values are expected to be represented using 32-bit limbs, and the procedure will fail if they are not. The stack transition looks as follows: [b_hi, b_lo, a_hi, a_lo, ...] -> [c, ...], where c = 1 when a == b, and 0 otherwise.
unchecked_eq	Performs equality comparison of two unsigned 64-bit integers. The input values are assumed to be represented using 32-bit limbs, but this is not checked. The stack transition looks as follows: [b_hi, b_lo, a_hi, a_lo, ...] -> [c, ...], where c = 1 when a == b, and 0 otherwise.
checked_neq	Performs inequality comparison of two unsigned 64-bit integers. The input values are expected to be represented using 32-bit limbs, and the procedure will fail if they are not. The stack transition looks as follows: [b_hi, b_lo, a_hi, a_lo, ...] -> [c, ...], where c = 1 when a != b, and 0 otherwise.
unchecked_neq	Performs inequality comparison of two unsigned 64-bit integers. The input values are assumed to be represented using 32-bit limbs, but this is not checked. The stack transition looks as follows: [b_hi, b_lo, a_hi, a_lo, ...] -> [c, ...], where c = 1 when a != b, and 0 otherwise.
checked_eqz	Performs comparison to zero of an unsigned 64-bit integer. The input value is assumed to be represented using 32-bit limbs, fails if it is not. The stack transition looks as follows: [a_hi, a_lo, ...] -> [c, ...], where c = 1 when a == 0, and 0 otherwise.
unchecked_eqz	Performs comparison to zero of an unsigned 64-bit integer. The input value is assumed to be represented using 32-bit limbs, but this is not checked. The stack transition looks as follows: [a_hi, a_lo, ...] -> [c, ...], where c = 1 when a == 0, and 0 otherwise.
checked_min	Compares two unsigned 64-bit integers and drop the larger one from the stack. The input values are expected to be represented using 32-bit limbs, and the procedure will fail if they are not. The stack transition looks as follows: [b_hi, b_lo, a_hi, a_lo, ...] -> [c_hi, c_lo, ...], where c = a when a < b, and b otherwise.
unchecked_min	Compares two unsigned 64-bit integers and drop the larger one from the stack. The input values are assumed to be represented using 32-bit limbs, but this is not checked. The stack transition looks as follows: [b_hi, b_lo, a_hi, a_lo, ...] -> [c_hi, c_lo, ...], where c = a when a < b, and b otherwise.
checked_max	Compares two unsigned 64-bit integers and drop the smaller one from the stack. The input values are expected to be represented using 32-bit limbs, and the procedure will fail if they are not. The stack transition looks as follows: [b_hi, b_lo, a_hi, a_lo, ...] -> [c_hi, c_lo, ...], where c = a when a > b, and b otherwise.
unchecked_max	Compares two unsigned 64-bit integers and drop the smaller one from the stack. The input values are assumed to be represented using 32-bit limbs, but this is not checked. The stack transition looks as follows: [b_hi, b_lo, a_hi, a_lo, ...] -> [c_hi, c_lo, ...], where c = a when a > b, and b otherwise.

Bitwise operations

Procedure	Description

checked_and	Performs bitwise AND of two unsigned 64-bit integers. The input values are assumed to be represented using 32-bit limbs, but this is not checked. The stack transition looks as follows: [[b_hi, b_lo, a_hi, a_lo, ...] -> [c_hi, c_lo, ...], where c = a AND b.
checked_or	Performs bitwise OR of two unsigned 64-bit integers. The input values are expected to be represented using 32-bit limbs, and the procedure will fail if they are not. The stack transition looks as follows: [b_hi, b_lo, a_hi, a_lo, ...] -> [c_hi, c_lo, ...], where c = a OR b.
checked_xor	Performs bitwise XOR of two unsigned 64-bit integers. The input values are expected to be represented using 32-bit limbs, and the procedure will fail if they are not. The stack transition looks as follows: [b_hi, b_lo, a_hi, a_lo, ...] -> [c_hi, c_lo, ...], where c = a XOR b.
overflowing_shl	Performs left shift of one unsigned 64-bit integer preserving the overflow and using the pow2 operation. The input value to be shifted is assumed to be represented using 32-bit limbs. The shift value should be in the range [0, 64), otherwise it will result in an error. The stack transition looks as follows: [b, a_hi, a_lo, ...] -> [d_hi, d_lo, c_hi, c_lo, ...], where (d,c) = a << b, which d contains the bits shifted out. This takes 35 cycles.
unchecked_shl	Performs left shift of one unsigned 64-bit integer using the pow2 operation. The input value to be shifted is assumed to be represented using 32-bit limbs. The shift value should be in the range [0, 64), otherwise it will result in an error. The stack transition looks as follows: [b, a_hi, a_lo, ...] -> [c_hi, c_lo, ...], where c = a << b mod 2^64. This takes 28 cycles.
overflowing_shr	Performs right shift of one unsigned 64-bit integer preserving the overflow and using the pow2 operation. The input value to be shifted is assumed to be represented using 32-bit limbs. The shift value should be in the range [0, 64), otherwise it will result in an error. The stack transition looks as follows: [b, a_hi, a_lo, ...] -> [d_hi, d_lo, c_hi, c_lo, ...], where c = a >> b, d = a << (64 - b). This takes 94 cycles.
unchecked_shr	Performs right shift of one unsigned 64-bit integer using the pow2 operation. The input value to be shifted is assumed to be represented using 32-bit limbs. The shift value should be in the range [0, 64), otherwise it will result in an error. The stack transition looks as follows: [b, a_hi, a_lo, ...] -> [c_hi, c_lo, ...], where c = a >> b. This takes 44 cycles.
unchecked_rotl	Performs left rotation of one unsigned 64-bit integer using the pow2 operation. The input value to be shifted is assumed to be represented using 32-bit limbs. The shift value should be in the range [0, 64), otherwise it will result in an error. The stack transition looks as follows: [b, a_hi, a_lo, ...] -> [c_hi, c_lo, ...], where c = a << b mod 2^64. This takes 35 cycles.
unchecked_rotr	Performs right rotation of one unsigned 64-bit integer using the pow2 operation. The input value to be shifted is assumed to be represented using 32-bit limbs. The shift value should be in the range [0, 64), otherwise it will result in an error. The stack transition looks as follows: [b, a_hi, a_lo, ...] -> [c_hi, c_lo, ...], where c = a << b mod 2^64. This takes 40 cycles.

---

id: mem title: Memory Procedures sidebar_label: Memory Procedures description: Mem Module contains a set of utility procedures for working with random access memory. keywords: - docs - polygon - wiki - miden - mem image: https://wiki.polygon.technology/img/thumbnail/polygon-miden.png

Module `std::mem` contains a set of utility procedures for working with random access memory.

Procedure	Description
memcopy	Copies `n` words from `read_ptr` to `write_ptr`. Stack transition looks as follows: [n, read_ptr, write_ptr, ...] -> [...] Cycles: 15 + 16n

---

id: sys title: System Procedures sidebar_label: System Procedures description: Sys module contains a set of system-level utility procedures. keywords: - docs - polygon - wiki - miden - system image: https://wiki.polygon.technology/img/thumbnail/polygon-miden.png

Module `std::sys` contains a set of system-level utility procedures.

Procedure	Description
truncate_stack	Removes elements deep in the stack until the depth of the stack is exactly 16. The elements are removed in such a way that the top 16 elements of the stack remain unchanged. If the stack would otherwise contain more than 16 elements at the end of execution, then adding a call to this function at the end will reduce the size of the public inputs that are shared with the verifier. Input: Stack with 16 or more elements. Output: Stack with only the original top 16 elements.
id: usage	
title: How to Use the Miden Virtual Machine	
sidebar_label: Use the Miden VM	
description: Basic instructions on how to use Miden VM.	
keywords:	

- docs
- polygon
- wiki
- miden
- cli
- interface image: https://wiki.polygon.technology/img/thumbnail/polygon-miden.png

---

Before you can use Miden VM, you'll need to make sure you have Rust installed. Miden VM v0.5 requires Rust version **1.67** or later.

Miden VM consists of several crates, each of which exposes a small set of functionality. The most notable of these crates are:

- miden-processor, which can be used to execute Miden VM programs.
- miden-prover, which can be used to execute Miden VM programs and generate proofs of their execution.
- miden-verifier, which can be used to verify proofs of program execution generated by Miden VM prover.

The above functionality is also exposed via the single miden crate, which also provides a CLI interface for interacting with Miden VM.

CLI interface

Compiling Miden VM

To compile Miden VM into a binary, we have a Makefile with the following tasks:

```
make exec
```

This will place an optimized, multi-threaded `miden` executable in the `./target/release` directory. It is equivalent to executing:

```
cargo build --profile optimized --features concurrent,executable
```

If you would like to enable single-threaded mode, you can compile Miden Vm using the following command:

```
cargo build --profile optimized --features executable
```

For a faster build, you can compile with less optimizations, replacing `--profile optimized` by `--release`. Example:

```
cargo build --release --features concurrent,executable
```

Controlling parallelism

Internally, Miden VM uses rayon for parallel computations. To control the number of threads used to generate a STARK proof, you can use `RAYON_NUM_THREADS` environment variable.

Running Miden VM

Once the executable has been compiled, you can run Miden VM like so:

```
./target/release/miden [subcommand] [parameters]
```

Currently, Miden VM can be executed with the following subcommands:

- `run` - this will execute a Miden assembly program and output the result, but will not generate a proof of execution.
- `prove` - this will execute a Miden assembly program, and will also generate a STARK proof of execution.
- `verify` - this will verify a previously generated proof of execution for a given program.
- `compile` - this will compile a Miden assembly program (i.e., build a program MAST) and outputs stats about the compilation process.
- `debug` - this will instantiate a CLI debugger against the specified Miden assembly program and inputs.
- `analyze` - this will run a Miden assembly program against specific inputs and will output stats about its execution.
- `repl` - this will initiate the Miden REPL tool.

All of the above subcommands require various parameters to be provided. To get more detailed help on what is needed for a given subcommand, you can run the following:

```
./target/release/miden [subcommand] --help
```

For example:

```
./target/release/miden prove --help
```

Fibonacci example

In the `miden/examples/fib` directory, we provide a very simple Fibonacci calculator example. This example computes the 1000th term of the Fibonacci sequence. You can execute this example on Miden VM like so:

```
./target/release/miden run -a miden/examples/fib/fib.masm -n 1
```

This will run the example code to completion and will output the top element remaining on the stack.

If you want the output of the program in a file, you can use the `--output` or `-o` flag and specify the path to the output file. For example:

```
./target/release/miden run -a miden/examples/fib/fib.masm -o fib.out
```

This will dump the output of the program into the `fib.out` file. The output file will contain the state of the stack at the end of the program execution.

Miden Development Tooling

Miden Debugger

The Miden debugger is a command-line interface (CLI) application, inspired on GNU gdb, that allows debugging of Miden assembly (MASM) programs. The debugger allows the user to step through the execution of the program, both forward and backward, either per clock cycle tick, or via breakpoints.

The Miden debugger supports the following commands:

Command	Shortcut	Arguments	Description
next	n	count?	Steps `count` clock cycles. Will step `1` cycle of `count` is ommitted.
continue	c	-	Executes the program until completion, failure or a breakpoint.
back	b	count?	Backward step `count` clock cycles. Will back-step `1` cycle of `count` is ommitted.
rewind	r	-	Executes the program backwards until the beginning, failure or a breakpoint.
print	p	-	Displays the complete state of the virtual machine.
print mem	p m	address?	Displays the memory value at `address`. If `address` is ommitted, didisplays all the memory values.
print stack	p s	index?	Displays the stack value at `index`. If `index` is ommitted, displays all the stack values.
clock	c	-	Displays the current clock cycle.
quit	q	-	Quits the debugger.
help	h	-	Displays the help message.

In order to start debugging, the user should provide a `MASM` program:

```
cargo run --features executable -- debug --assembly miden/examples/nprime/nprime.masm
```

The expected output is:

```
============================================================
Debug program
============================================================
Reading program file `miden/examples/nprime/nprime.masm`
Compiling program... done (16 ms)
Debugging program with hash 11dbbddff27e26e48be3198133df8cbed6c5875d0fb
606c9f037c7893fde4118...
Reading input file `miden/examples/nprime/nprime.inputs`
Welcome! Enter `h` for help.
>>
```

In order to add a breakpoint, the user should insert a `breakpoint` instruction into the MASM file. This will generate a `Noop` operation that will be decorated with the debug break configuration. This is a provisory solution until the source mapping is implemented.

The following example will halt on the third instruction of `foo`:

```
proc.foo
    dup
    dup.2
    breakpoint
    swap
    add.1
end

begin
    exec.foo
end
```

REPL

The Miden Read–eval–print loop (REPL) is a Miden shell that allows for quick and easy debugging of Miden assembly. After the REPL gets initialized, you can execute any Miden instruction, undo executed instructions, check the state of the stack and memory at a given point, and do many other useful things! When the REPL is exited, a `history.txt` file is saved. One thing to note is that all the REPL native commands start with an `!` to differentiate them from regular assembly instructions. The REPL currently supports the following commands:

Miden assembly instruction

All Miden instructions mentioned in the Miden Assembly sections are valid. One can either input instructions one by one or multiple instructions in one input.

For example, the below two commands will result in the same output.

```
>> push.1
>> push.2
>> push.3
```

```
push.1 push.2 push.3
```

To execute a control flow operation, one must write the entire statement in a single line with spaces between individual operations.

```
repeat.20
    pow2
end
```

The above example should be written as follows in the REPL tool:

```
repeat.20 pow2 end
```

!help

The `!help` command prints out all the available commands in the REPL tool.

!program

The `!program` command prints out the entire Miden program being executed. E.g., in the below scenario:

```
>> push.1.2.3.4
>> repeat.16 pow2 end
>> u32checked_add

>> !program
begin
    push.1.2.3.4
    repeat.16 pow2 end
    u32checked_add
end
```

!stack

The `!stack` command prints out the state of the stack at the last executed instruction. Since the stack always contains at least 16 elements, 16 or more elements will be printed out (even if all of them are zeros).

```
>> push.1 push.2 push.3 push.4 push.5
>> exp
>> u32checked_mul
>> swap
>> eq.2
>> assert
```

The `!stack` command will print out the following state of the stack:

```
>> !stack
3072 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

!mem

The `!mem` command prints out the contents of all initialized memory locations. For each such location, the address, along with its memory values, is printed. Recall that four elements are stored at each memory address.

If the memory has at least one value that has been initialized:

```
>> !mem
7: [1, 2, 0, 3]
8: [5, 7, 3, 32]
9: [9, 10, 2, 0]
```

If the memory is not yet been initialized:

```
>> !mem
The memory has not been initialized yet
```

!mem[addr]

The `!mem[addr]` command prints out memory contents at the address specified by `addr`.

If the `addr` has been initialized:

```
>> !mem[9]
9: [9, 10, 2, 0]
```

If the `addr` has not been initialized:

```
>> !mem[87]
Memory at address 87 is empty
```

!undo

The `!undo` command reverts to the previous state of the stack and memory by dropping off the last executed assembly instruction from the program. One could use `!undo` as often as they want to restore the state of a stack and memory n instructions ago (provided there are n instructions in the program). The `!undo` command will result in an error if no remaining instructions are left in the Miden program.

```
>> push.1 push.2 push.3
>> push.4
>> !stack
4 3 2 1 0 0 0 0 0 0 0 0 0 0 0 0

>> push.5
>> !stack
5 4 3 2 1 0 0 0 0 0 0 0 0 0 0 0

>> !undo
4 3 2 1 0 0 0 0 0 0 0 0 0 0 0 0

>> !undo
3 2 1 0 0 0 0 0 0 0 0 0 0 0 0 0
```

---

id: alchemy title: How to Deploy a Smart Contract Using Alchemy sidebar_label: How to Deploy a Smart Contract Using Alchemy description: Guide to deploy smart contracts using Alchemy keywords: - docs - matic - polygon - alchemy - create smart contract - deploy on polygon image: https://wiki.polygon.technology/img/polygon-logo.png

:::caution Content disclaimer

Please view the third-party content disclaimer here.

:::

Overview

This tutorial is for developers who are either new to Ethereum blockchain development or want to understand the fundamentals of deploying and interacting with smart contracts. It will walk you through creating and deploying a smart contract on the Polygon Mumbai test network using a cryptocurrency wallet (Metamask), Solidity, Hardhat, and Alchemy.

:::tip

If you have questions or concerns, please reach out to the Alchemy team via their official Discord server.

:::

What you will learn

To create a smart contract in this tutorial, you will learn how to use Alchemy's platform to:

- Create a smart contract application
- Check a wallet's balance
- Verify contract calls in a blockchain explorer

What you will do

Following the tutorial, you will:

1. Get started creating an app on Alchemy
2. Create a wallet address with Metamask
3. Add balance to the wallet (using test tokens)
4. Use Hardhat and Ethers.js to compile and deploy the project
5. Check contract status on Alchemy's platform

Create and Deploy your Smart Contract

Connect to the Polygon network

There are several ways to make requests to the Polygon PoS chain. Rather than running your own node, you will use a free account on Alchemy's developer platform and interact with the Alchemy Polygon PoS API to communicate with the Polygon PoS chain. The platform consists of a full suite of developer tooling – this includes the ability to monitor requests, data analytics that demonstrates what happens under the hood during smart contract deployment, enhanced APIs (Transact, NFTs, etc), and an ethers.js SDK.

If you don't already have an Alchemy account, start by signing up for a free account here. After creating your account, you have the option of immediately creating your first app before reaching the dashboard.

img

Create your App (and API key)

After successfully creating an Alchemy account, you will need to generate an API key by creating an app. This authenticates the requests made to the Polygon Mumbai testnet. If you're not familiar with testnets, check out this testnet guide.

To generate a new API key, navigate to the **Apps** tab on the Alchemy dashboard navigation bar and select the **Create App** sub-tab.

img

Name your new app **Hello World**, offer a short description, select **Polygon** for the chain, and choose **Polygon Mumbai** for your network.

Finally, click on **Create app**. Your new app should appear in the table below.

Create a Wallet address

Polygon PoS is a layer 2 scaling solution for Ethereum. Therefore, we need an Ethereum wallet and add a custom Polygon URL to send and receive transactions on the Polygon Mumbai testnet. For this tutorial, we will use MetaMask, a browser-compatible cryptocurrency wallet used to manage your wallet address. If you want to understand more about how transactions on Ethereum work, check out this transactions guide by the Ethereum Foundation.

To get your custom Polygon RPC URL from Alchemy, go to your **Hello World** app in your Alchemy dashboard and click **View Key** in the top right corner. Then go ahead and copy your Alchemy HTTP API key.

img

You can download and create a MetaMask account for free here. Once you've created an account, follow these steps to set up the Polygon PoS network on your wallet.

1. Select **Settings** from the drop-down menu in the top right corner of your MetaMask wallet.

2. Select **Networks** from the menu to the left.

3. Connect your wallet to the Mumbai Testnet using the following parameters:

   Network Name: Polygon Mumbai Testnet

   New RPC URL: https://polygon-mumbai.g.alchemy.com/v2/your-api-key

   ChainID: 80001

   Symbol: MATIC

   Block Explorer URL: https://mumbai.polygonscan.com/

Add Polygon Mumbai Test MATIC

You'll need a few testnet tokens to deploy your smart contract to the Mumbai testnet. To obtain testnet tokens, go to the Polygon Mumbai Faucet, select **Mumbai**, select **MATIC Token**, and enter your Polygon wallet address, then click **Submit**. Due to network traffic, it may take some time to receive your testnet tokens.

You can also use Alchemy's free Mumbai faucet.

img

You will see the testnet tokens in your MetaMask account soon after.

Check your Wallet Balance

To double-check our balance is there, let's make an eth_getBalance request using Alchemy's composer tool. Select **Polygon** as the chain, **Polygon Mumbai** as the network, `eth_getBalance` as the method, and input your address. This will return the amount of MATIC in our wallet. Check out this video for instructions on how to use the composer tool.

img

After you input your MetaMask account address and click **Send Request**, you should see a response that looks like this:

```
{ "jsonrpc": "2.0", "id": 0, "result": "0xde0b6b3a7640000" }
```

:::info

This result is in Wei, not ETH. Wei is the smallest denomination of Ether. The conversion from Wei to Ether is: 1 Ether = 10^18 Wei. So, if we convert "0xde0b6b3a7640000" to decimal, we get 1*10^18, which equals 1 ETH. This can be mapped to 1 MATIC based on denomination.

:::

Initialize your project

First, we'll need to create a folder for our project. Navigate to your command line and type:

```
mkdir hello-world
cd hello-world
```

Now that we're inside our project folder, we'll use `npm init` to initialize the project. If you don't already have npm installed, follow these instructions (we'll also need Node.js so download that too!).

```
npm init # (or npm init --yes)
```

It doesn't really matter how you answer the installation questions, here is how we did it for reference:

```
package name: (hello-world)
version: (1.0.0)
description: hello world smart contract
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)

About to write to /Users/.../.../.../hello-world/package.json:

{
   "name": "hello-world",
   "version": "1.0.0",
   "description": "hello world smart contract",
   "main": "index.js",
   "scripts": {
      "test": "echo \"Error: no test specified\" && exit 1"
   },
   "author": "",
   "license": "ISC"
}
```

Approve the package.json and we're good to go!

Download Hardhat

Hardhat is a development environment to compile, deploy, test, and debug your Ethereum software. It helps developers when building smart contracts and dApps locally before deploying to the live chain.

Inside our `hello-world` project, run:

```
npm install --save-dev hardhat
```

Check out this page for more details on installation instructions.

Create Hardhat project

Inside our `hello-world` project folder, run:

```
npx hardhat
```

You should see a welcome message and option to select what you want to do. Select **create an empty hardhat.config.js**:

```
888    888                      888 888               888
888    888                      888 888               888
888    888                      888 888               888
8888888888  8888b.  888d888 .d88888 88888b.   8888b.  888888
888    888     "88b 888P"  d88" 888 888 "88b     "88b 888
888    888 .d888888 888    888  888 888  888 .d888888 888
888    888 888  888 888    Y88b 888 888  888 888  888 Y88b.
888    888 "Y888888 888     "Y88888 888  888 "Y888888  "Y888

👷 Welcome to Hardhat v2.0.11 👷

What do you want to do? …
Create a sample project
❯ Create an empty hardhat.config.js
Quit
```

This will generate a `hardhat.config.js` file for us, which is where we'll specify all of the set up for our project.

Add project folders

To keep our project organized, we'll create two new folders. Navigate to the root directory of your `hello-world` project in your command line and type:

```
mkdir contracts
mkdir scripts
```

- `contracts/` is where we'll keep our hello world smart contract code file
- `scripts/` is where we'll keep scripts to deploy and interact with our contract

Write the contract

Open up the **hello-world** project in your favorite editor, such as VSCode. Smart contracts are written in a language called Solidity which is what we will use to write our `HelloWorld.sol` smart contract.

1. Navigate to the `contracts` folder and create a new file called `HelloWorld.sol`
2. Below is a sample Hello World smart contract from the Ethereum Foundation that we will be using for this tutorial. Copy and paste in the contents below into your `HelloWorld.sol` file, and be sure to read the comments to understand what this contract does:

```
// SPDX-License-Identifier: None

// Specifies the version of Solidity, using semantic versioning.
// Learn more: https://solidity.readthedocs.io/en/v0.5.10/layout-of-source-files.html#pragma
pragma solidity >=0.8.9;

// Defines a contract named `HelloWorld`.
// A contract is a collection of functions and data (its state). Once deployed, a contract resides at a specific address on the
Ethereum blockchain. Learn more: https://solidity.readthedocs.io/en/v0.5.10/structure-of-a-contract.html
contract HelloWorld {

   //Emitted when update function is called
   //Smart contract events are a way for your contract to communicate that something happened on the blockchain to your app
front-end, which can be 'listening' for certain events and take action when they happen.
   event UpdatedMessages(string oldStr, string newStr);

   // Declares a state variable `message` of type `string`.
   // State variables are variables whose values are permanently stored in contract storage. The keyword `public` makes
variables accessible from outside a contract and creates a function that other contracts or clients can call to access the
value.
   string public message;

   // Similar to many class-based object-oriented languages, a constructor is a special function that is only executed upon
contract creation.
   // Constructors are used to initialize the contract's data. Learn
more:https://solidity.readthedocs.io/en/v0.5.10/contracts.html#constructors
   constructor(string memory initMessage) {

      // Accepts a string argument `initMessage` and sets the value into the contract's `message` storage variable).
      message = initMessage;
   }

   // A public function that accepts a string argument and updates the `message` storage variable.
```

```
    function update(string memory newMessage) public {
       string memory oldMsg = message;
       message = newMessage;
       emit UpdatedMessages(oldMsg, newMessage);
    }
}
```

This is a super simple smart contract that stores a message upon creation and can be updated by calling the `update` function.

Connect with MetaMask & Alchemy

We've created a MetaMask wallet, Alchemy account, and written our smart contract, now it's time to connect the three.

Every transaction sent from your virtual wallet requires a signature using your unique private key. To provide our program with this permission, we can safely store our private key (and Alchemy API key) in an environment file.

First, install the dotenv package in your project directory:

```
npm install dotenv --save
```

Then, create a `.env` file in the root directory of our project, and add your MetaMask private key and HTTP Alchemy API URL to it.

:::warning Warning

Your environment file must be named `.env` or it won't be recognized as an environment file. Do not name it `process.env` or `.env-custom` or anything else.

Also, if you are using a version control system like git to manage your project, please **DO NOT** track the `.env` file. Add `.env` to your `.gitignore` file to avoid publishing secret data.

:::

- Follow these instructions to export your private key
- To get your Alchemy HTTP API key (RPC URL), navigate to your **Hello World** app on your account's dashboard and click **View Key** in the top right corner.

Your `.env` should look like this:

```
API_URL = "https://polygon-mumbai.g.alchemy.com/v2/your-api-key"
PRIVATE_KEY = "your-metamask-private-key"
```

To actually connect these to our code, we'll reference these variables in our `hardhat.config.js` file later in this tutorial.

Install Ethers.js

Ethers.js is a library that makes it easier to interact and make requests to Ethereum by wrapping standard JSON-RPC methods with more user-friendly methods.

Hardhat makes it easy to integrate plugins for additional tooling and extended functionality. We'll be taking advantage of the Ethers plugin for contract deployment. Ethers.js has useful contract deployment methods.

In your project directory, type:

```
npm install --save-dev @nomiclabs/hardhat-ethers "ethers@^5.0.0"
```

We'll also require ethers in our `hardhat.config.js` in the next step.

Update hardhat.config.js

We've added several dependencies and plugins so far. Now we need to update `hardhat.config.js` so that our project recognizes those dependencies.

Update your `hardhat.config.js` to look like this:

```
/**
* @type import('hardhat/config').HardhatUserConfig
*/

require('dotenv').config();
require("@nomiclabs/hardhat-ethers");

const { API_URL, PRIVATE_KEY } = process.env;

module.exports = {
   solidity: "0.8.9",
   defaultNetwork: "polygon_mumbai",
   networks: {
      hardhat: {},
      polygon_mumbai: {
         url: API_URL,
         accounts: [`0x${PRIVATE_KEY}`]
      }
   },
}
```

Compile our Smart Contract

To make sure everything is working so far, let's compile our contract. The `compile` task is one of the built-in hardhat tasks.

From the command line run:

```
npx hardhat compile
```

You might get a warning about `SPDX license identifier not provided in source file` , but the application might still be working fine. If not, you can always message in the Alchemy discord.

Write our deploy script

Now that our contract is written and our configuration file is good to go, it's time to write our contract deploy script.

Navigate to the `scripts/` folder and create a new file called `deploy.js`, adding the following contents to it:

```
async function main() {
   const HelloWorld = await ethers.getContractFactory("HelloWorld");

   // Start deployment, returning a promise that resolves to a contract object
   const hello_world = await HelloWorld.deploy("Hello World!");
   console.log("Contract deployed to address:", hello_world.address);
}

main()
  .then(() => process.exit(0))
  .catch(error => {
    console.error(error);
    process.exit(1);
  });
```

We've adopted the Hardhat team explanations to what each of these lines of code does from their Contracts tutorial here.

```
const HelloWorld = await ethers.getContractFactory("HelloWorld");
```

A `ContractFactory` in ethers.js is an abstraction used to deploy new smart contracts, so `HelloWorld` here is a factory for instances of our hello world contract. When using the `hardhat-ethers` plugin `ContractFactory` and `Contract` , instances are connected to the first signer (owner) by default.

```
const hello_world = await HelloWorld.deploy();
```

Calling `deploy()` on a `ContractFactory` will start the deployment, and return a `Promise` that resolves to a `Contract` object. This is the object that has a method for each of our smart contract functions.

Deploy our Smart Contract

Navigate to the command line and run:

```
npx hardhat run scripts/deploy.js --network polygon_mumbai
```

You should see something like this:

```
Contract deployed to address: 0x3d94af870ED272Cd5370e4135F9B2Bd0e311d65D
```

If we go to the Polygon Mumbai explorer and search for our contract address, we should be able to see that it has been deployed successfully.

The `From` address should match your MetaMask account address and the `To` address will say **Contract Creation**. But if we click into the transaction, we'll see our contract address in the `To` field.

img

Verify the Contract

Alchemy provides an explorer where you can find information about the methods deployed along with the smart contract, such as response time, HTTP status, error codes among others. It is a good environment to verify your contract and check whether the transactions went through.

img

Congratulations! You just deployed a smart contract to the Polygon Mumbai network.

Additional Resources

- How to Develop an NFT Smart Contract – Alchemy has a written tutorial with a Youtube video on this topic. This is week 1 of its free 10 week **Road to Web3** dev series

- Polygon API Quickstart – Alchemy's developer docs guide to getting up and running with Polygon

---

id: chainide title: How to Deploy a Smart Contract Using ChainIDE sidebar_label: How to Deploy a Smart Contract Using ChainIDE description: Learn how to use ChainIDE to build and deploy smart contracts. keywords: - docs - matic - polygon - IDE - build - deploy - environment - smart contract - chainIDE image: https://wiki.polygon.technology/img/polygon-logo.png

:::caution Content disclaimer

Please view the third-party content disclaimer here.

:::

Introduction

ChainIDE is a chain agnostic, cloud-based IDE for creating decentralized applications. It enhances development cycle through pre-configured plugins that save users' time and effort. This is a beginner guide on creating and deploying a simple ERC-721 smart contract on the Polygon Mumbai Testnet.

:::tip

If you have any questions, feel free to ask the ChainIDE team in their Discord.

:::

Prerequisites

1. ChainIDE
2. MetaMask
3. Solidity

What you will do

The following are general steps for deploying an ERC-721 smart contract:

1. Set up a wallet
2. Write down an ERC-721 smart contract
3. Compile an ERC-721 Smart Contract
4. Deploy an ERC-721 Smart Contract
5. Create a Flattened File using Flattener Library
6. Verify a Smart Contract
7. NFT Minting

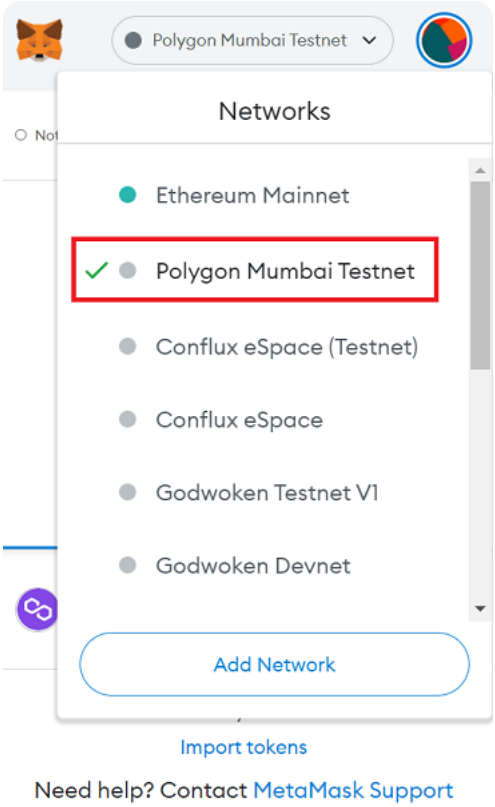
Setting up a Wallet

Install MetaMask

A gas fee must be paid when deploying a smart contract on the blockchain or making a transaction to a deployed smart contract, and we must do so using a crypto wallet, such as MetaMask. To download MetaMask, click here.

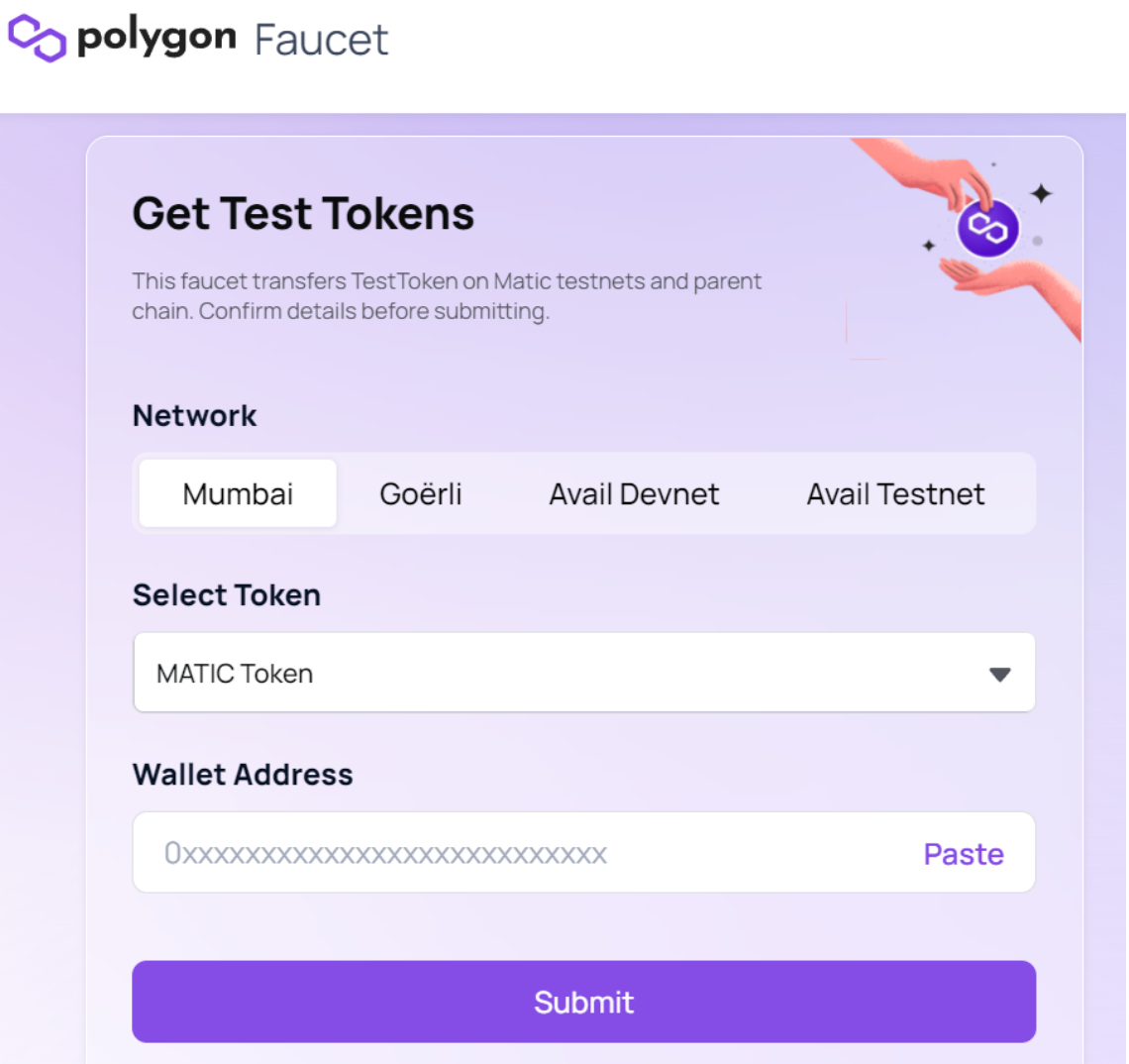
Adding the Mumbai testnet

After installing MetaMask, you need to add the Polygon Mumbai Testnet to MetaMask. In order to add Mumbai Testnet, check out the configuration guide for MetaMask.



Obtaining MATIC tokens on Testnet

After adding the Mumbai testnet on MetaMask, visit the Polygon Faucet to request testnet tokens. These tokens are required to pay gas fees in order to deploy and interact with the smart contract. On the faucet page, select **Mumbai** as the network, **MATIC** as the token, and paste your MetaMask wallet address. Then, click **Submit** and the faucet will send you some test MATIC within a minute.



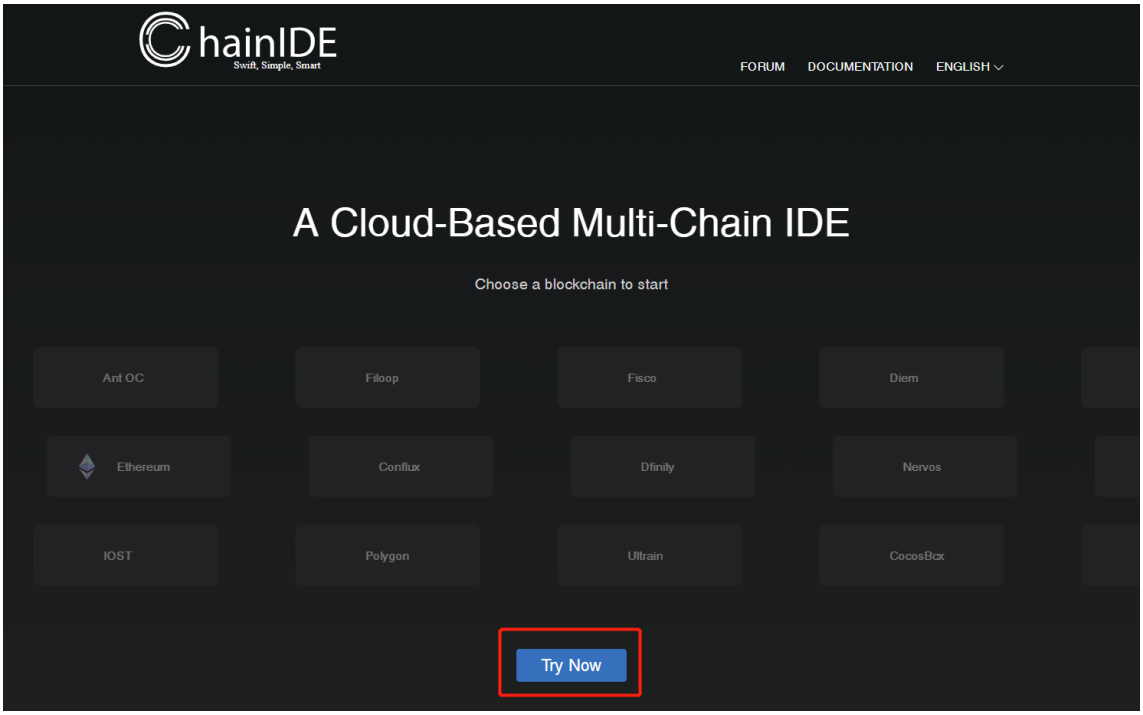
Write an ERC-721 Smart Contract

You need to write down all the required functions that you want to implement in your ERC-721 smart contract. A general ERC-721 smart contract has the following functions:

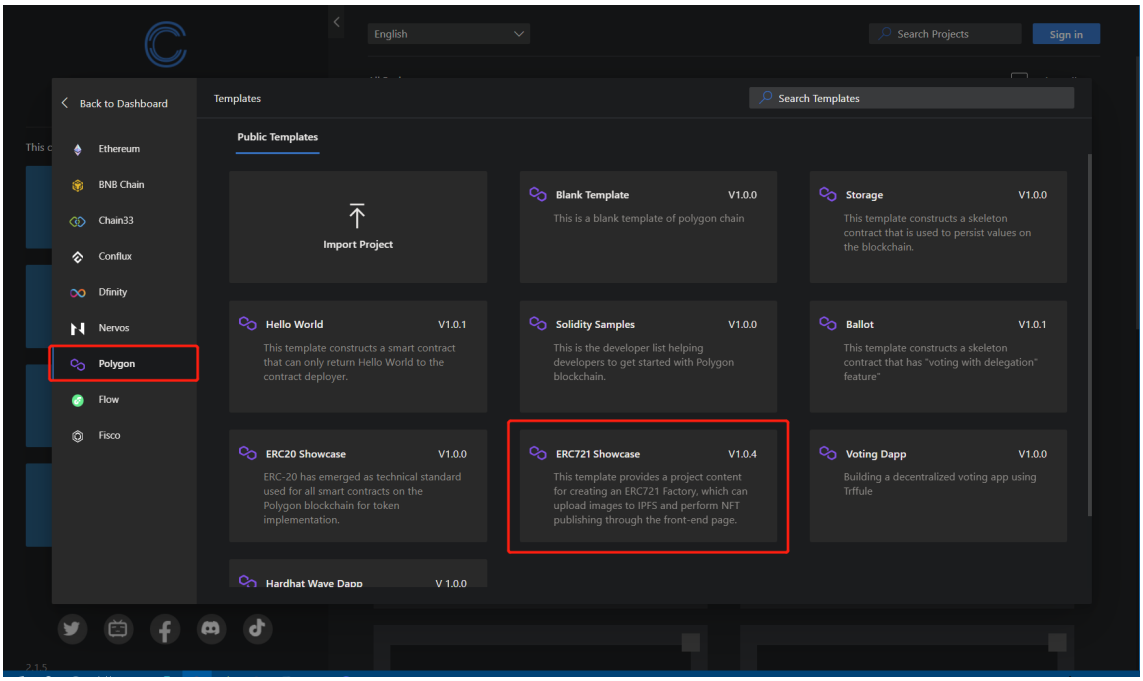
- `balanceOf()` : returns the number of NFTs held by the owner.
- `ownerOf()` : returns the address of the token holder.
- `approve()` : approves the permission to transfer tokens on the owner's behalf. Approval event needs to be triggered after the method is successful.
- `setApprovalForAll()` : Sets the approval of a given operator, and the `approvalforall` event needs to be triggered after success.
- `getApproved()` : Get the approved address for a single NFT.
- `isApprovedForAll()` : Query if an address is an authorized operator for another address.
- `safeTransferFrom()` : To transfer the ownership of an NFT, a successful transfer operation must initiate a transfer event.
- `transferFrom()` : Used to transfer NFTs. After the method succeeds, it needs to trigger the transfer event. The caller confirms he can receive NFT normally, otherwise, this NFT will be lost. When this function is implemented, it needs to check whether it meets the judgment conditions.

The ChainIDE team has prepared a complete ERC-721 template contract that includes all the required functions; you may use this built-in template and add/delete functions according to your requirements.

Visit the ChainIDE site and click on **Try Now**.

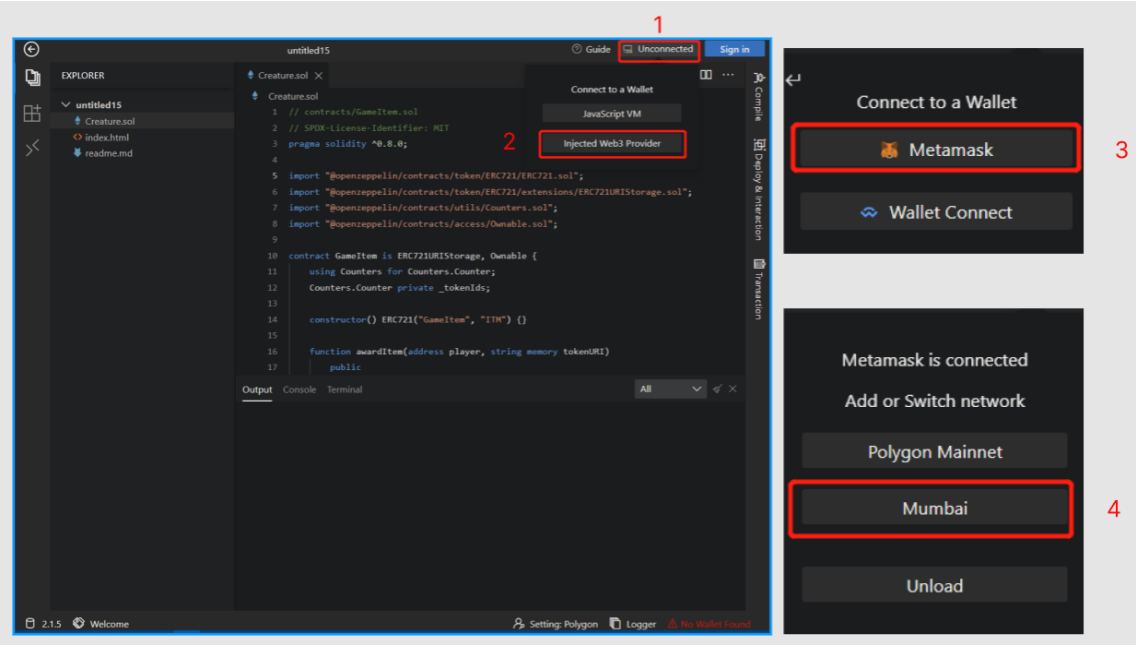


Then, click on **New Project** and select **Polygon**. You will be presented with a list of available public templates. Select **ERC721 Showcase**.



Now, you can see the template contract, `Creature.sol`, that includes all the required functions.

After creating the project, click on the **Unconnected** button in the upper right corner, select the **Injected Web3 Provider** button, and then click on MetaMask to connect your wallet (Polygon Mainnet is the main network, and Mumbai is the test network - connect to Mumbai).



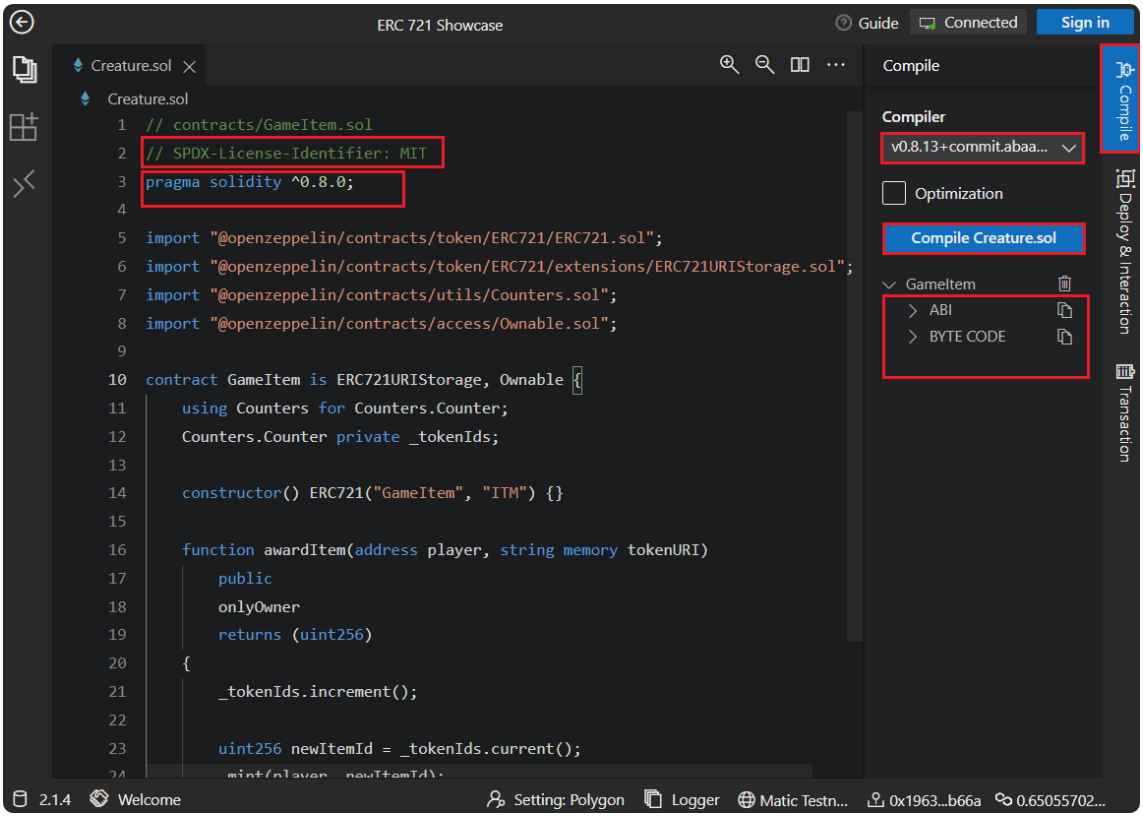
Compile an ERC-721 Smart Contract

After you have completed your smart contract, it is time to compile it. To compile, navigate to "compile the module", choose an appropriate compiler according to your source code, and press the "compile" button. An ABI and bytecode for the source code are generated upon successful compilation. If there are any errors in your source code, they will be displayed in the "Logger module" under the output panel. You may need to carefully read the error, resolve it, and recompile the contract.

:::note

Note down the **compiler version** and the **license** for your source code. It would be needed when you **Verify** your smart contract on the **Polygon Mumbai Testnet**.

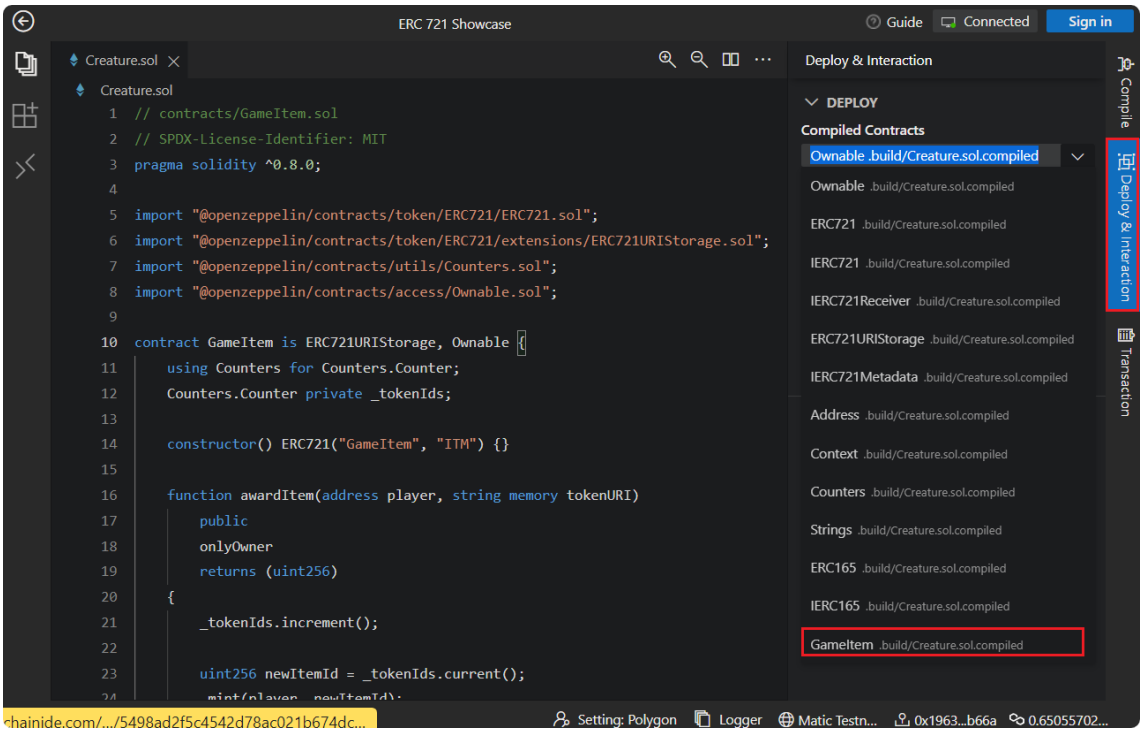
:::



Deploy an ERC-721 Smart Contract

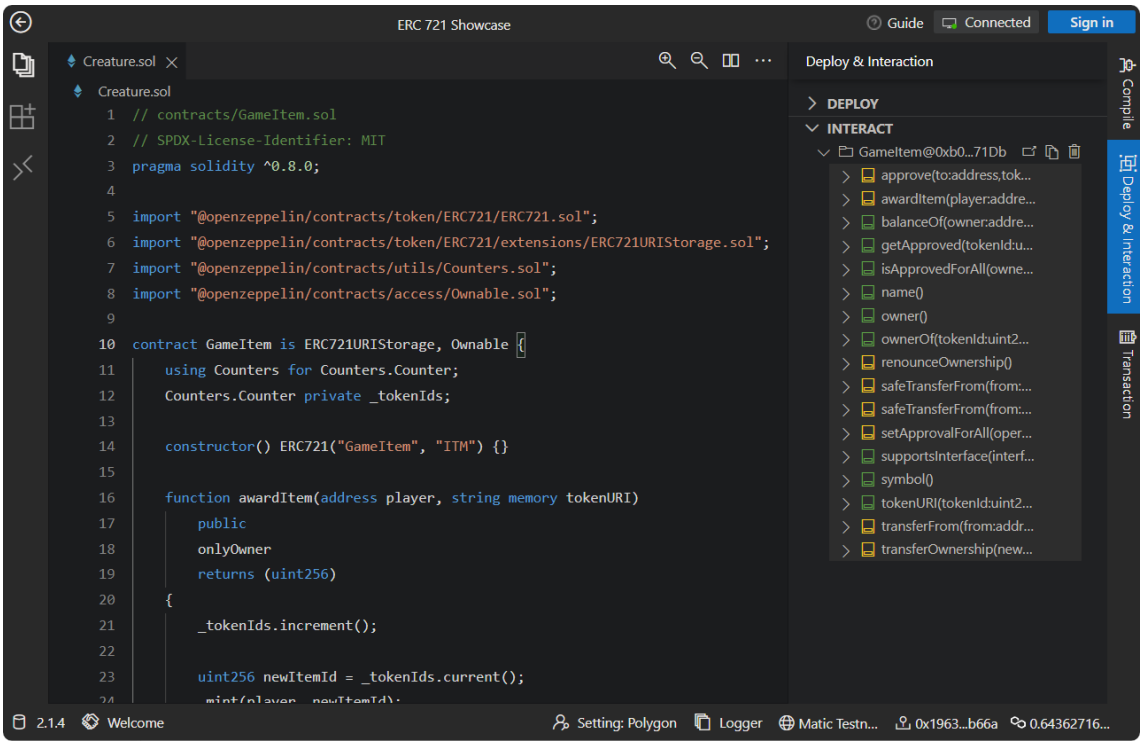
After successful compilation, it's time to deploy your compiled ERC-721 smart contract to the Polygon Mumbai Testnet. Before you deploy, you need to have MetaMask installed, the Mumbai test network added to your wallet, and some testnet MATIC tokens to pay for the transaction fees.

Navigate to the **Deploy & Interaction** module and choose among the compiled smart contract. Select the smart contract you want to deploy and click the **Deploy** button. For this tutorial, the `GameItem` smart contract will be deployed.



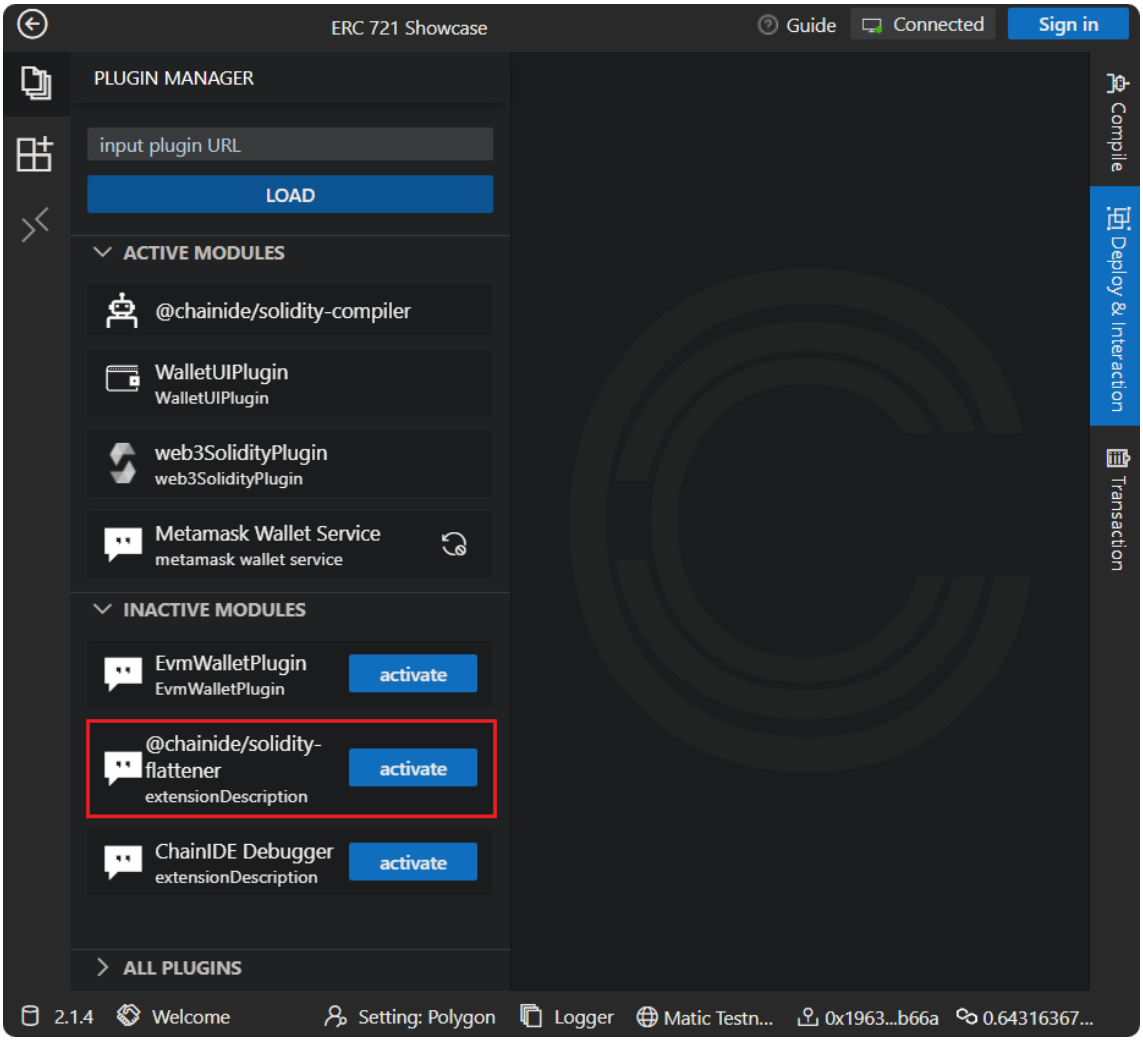
Check Functions from the Deployed Contract

After successful deployment, an output message should state that your smart contract was deployed successfully. You can now verify the deployed contract. All the functions in the deployed smart contract can be seen in the **INTERACT** panel.

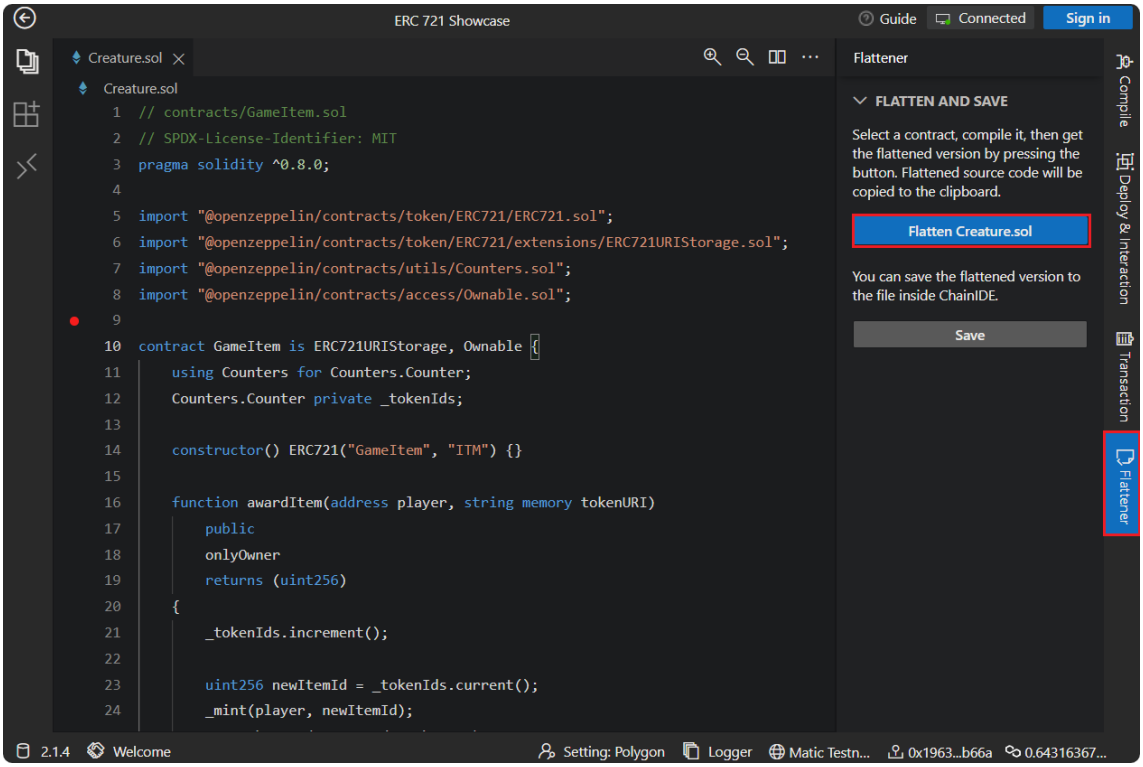


Create a Flattened File using Flattener Library

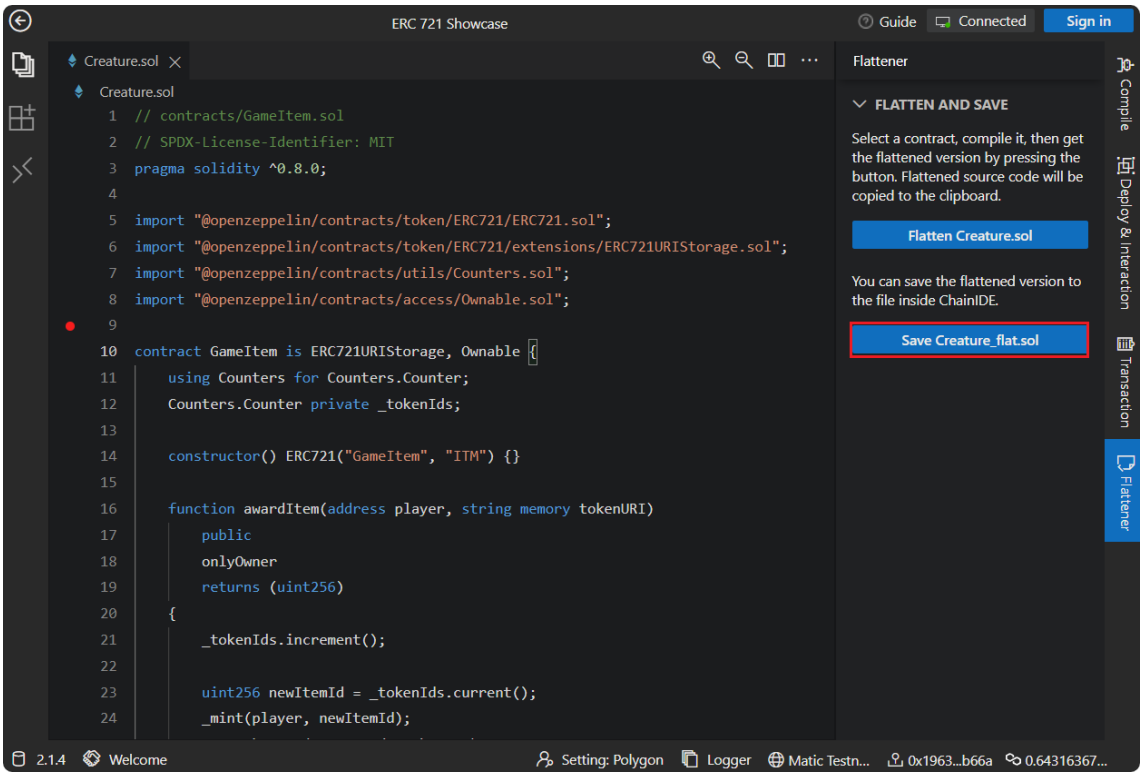
To verify a smart contract that imports other smart contracts, we need to create a flattened file. A flattened file includes all the source codes of imported contracts in a single file. To create a flattened file, you need to add a **Flattener** plug-in.



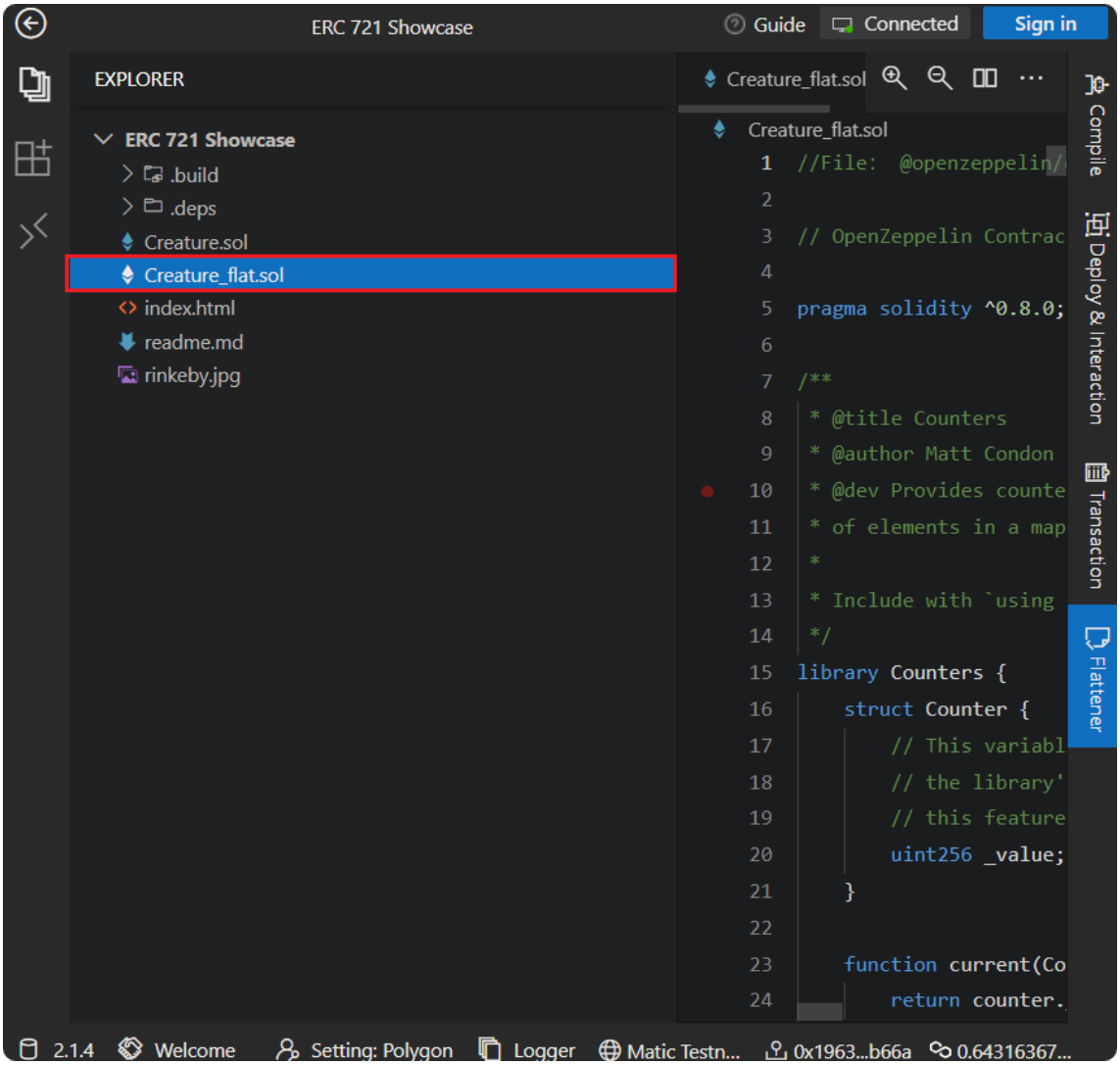
Once the **Flattener** plug-in is activated, you'll be able to access it as a separate module as shown in the figure below. Choose the compiled file, and click on the **Flatten** button to create a flattened file. Once the flattened file is created, it will be automatically copied to the clipboard. You may paste it into a file and save it for later usage.



If you want to save the flattened file, click the **Save** button, and a flattened file will be saved in the current repository.

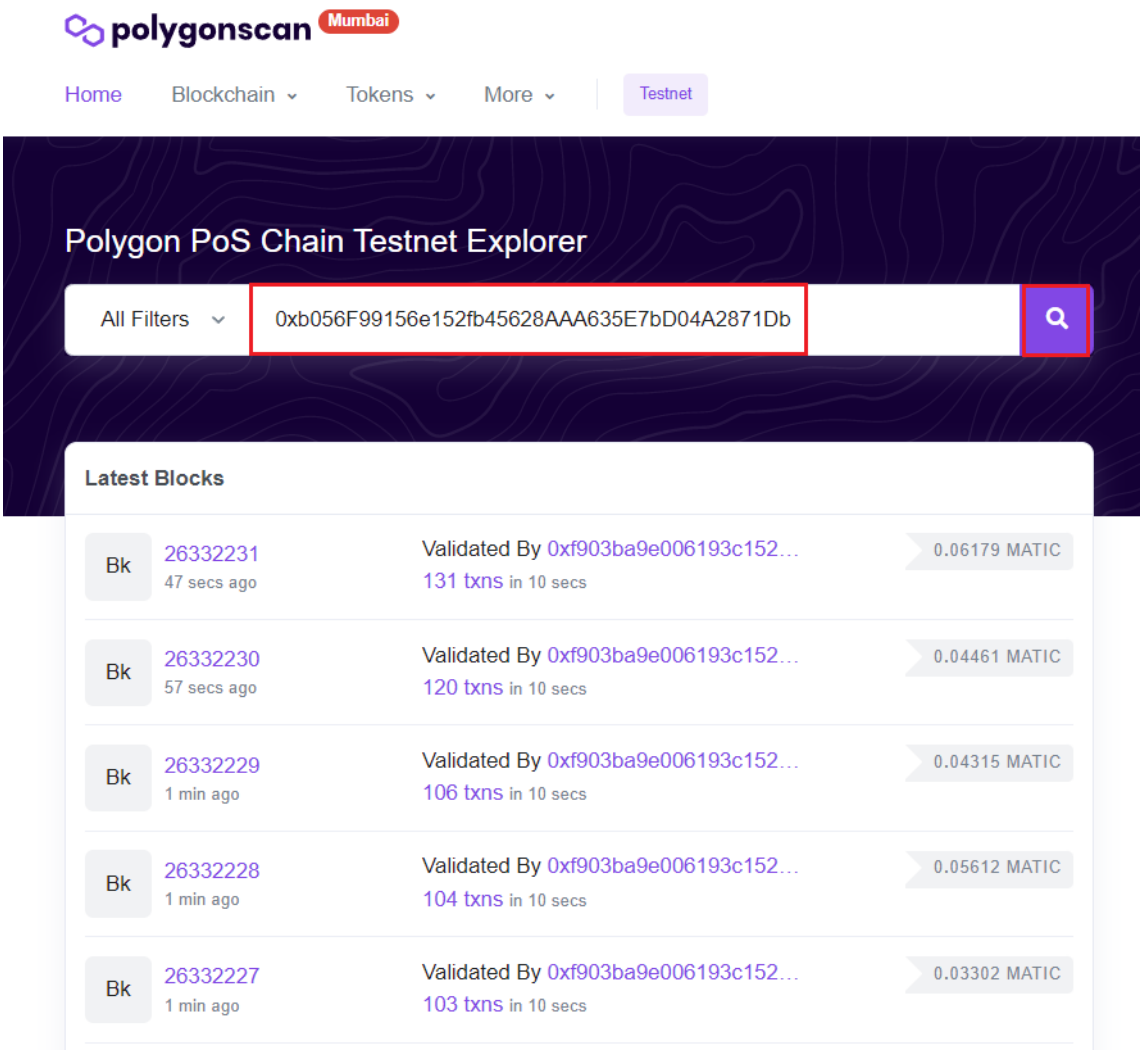


The saved flattened file can be accessed under the explorer module.

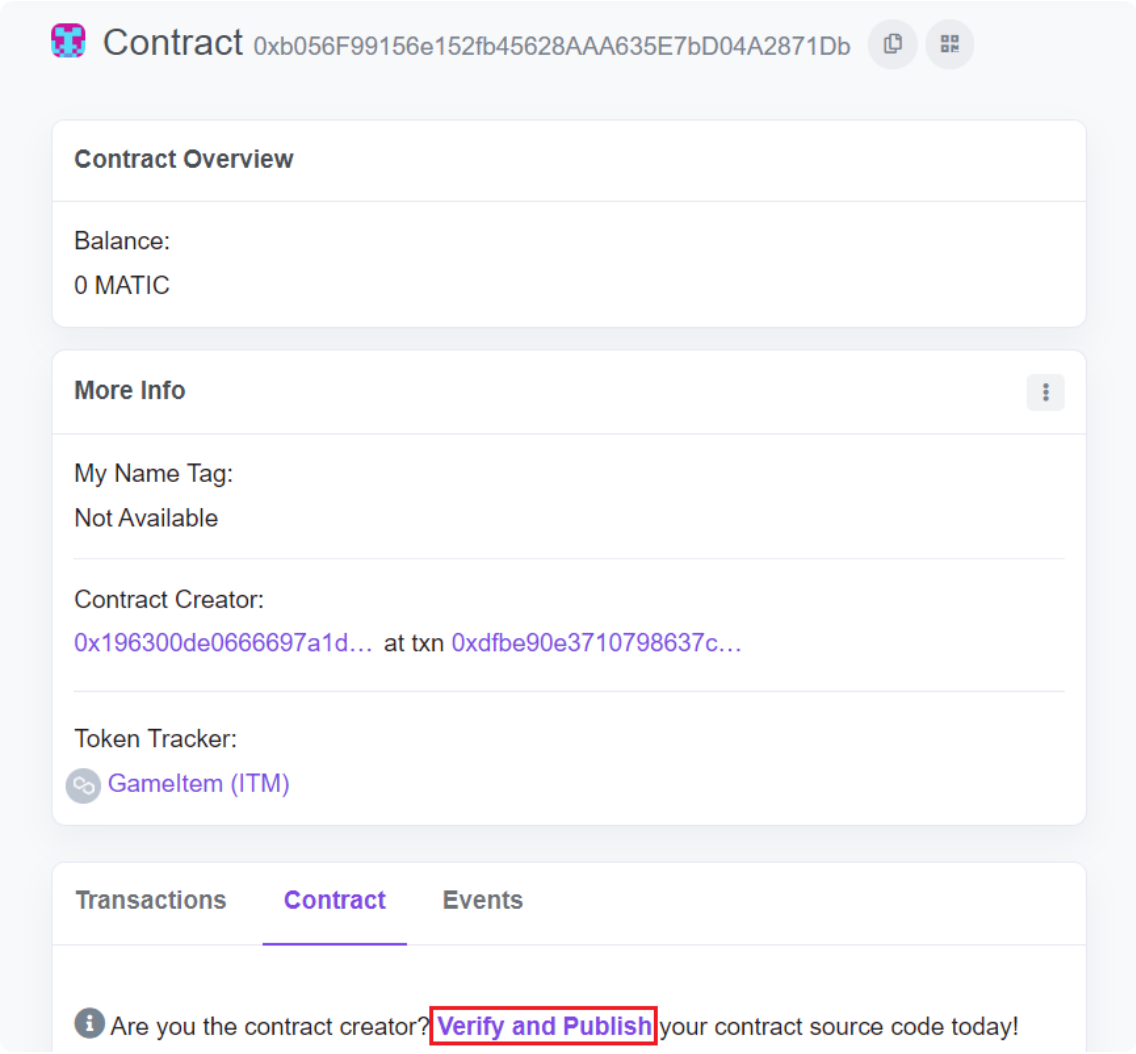


Verify a Smart Contract

To verify a smart contract, you need to visit Mumbai Polygonscan, and search for the deployed smart contract using the contract address.



Click on the **Verify and Publish** link shown under the contract section.



Once you click on the verify and publish link, you will be asked for the following:

- Contract Address: The address of a deployed smart contract that you want to verify
- Compiler Type: Either you want to verify a single file or multiple files
- Compiler Version: The compiler version that you used to compile the smart contract
- License: Open-source license type that you used for your source code

Verify & Publish Contract Source Code

COMPILER TYPE AND VERSION SELECTION

Source code verification provides **transparency** for users interacting with smart contracts. By uploading the source code, PolygonScan will match the compiled code with that on the blockchain. Just like contracts, a "smart contract" should provide end users with more information on what they are "digitally signing" for and give users an opportunity to audit the code to independently verify that it actually does what it is supposed to do.

Please enter the Contract Address you would like to verify

0xb056F99156e152fb45628AAA635E7bD04A2871Db

Please select Compiler Type

Solidity (Single file)

Please select Compiler Version

v0.8.13+commit.abaa5c0e

☑ Un-Check to show all nightly Commits also

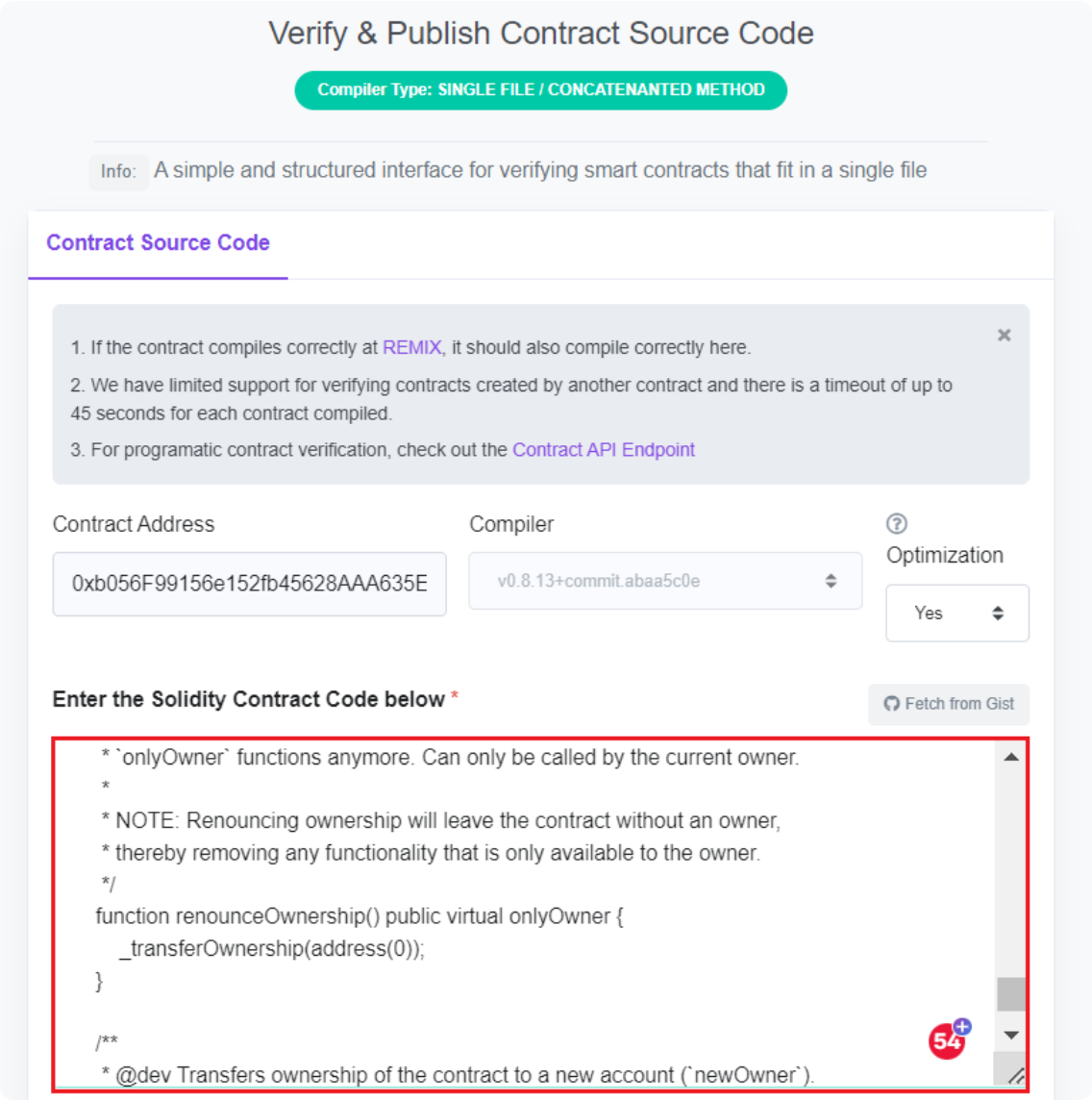
Please select Open Source License Type

3) MIT License (MIT)

☑ I agree to the terms of service

Continue Reset

After that, you need to paste the flattened file that you created in step 5, and your smart contract will be verified.



If there are no issues with your smart contract, it would be verified, and you'll be able to see an image similar to the one that is shown below.

Contract 0xb056F99156e152fb45628AAA635E7bD04A2871Db

Contract Overview

Balance: 0 MATIC

More Info

My Name Tag: Not Available

Creator: 0x196300de0666697a1d... at txn 0xdfbe90e3710798637c...

Tracker: GameItem (ITM)

Transactions | Contract | Events

Code | Read Contract | Write Contract | Search Source Code

Contract Source Code Verified (Exact Match)

Contract Name: **GameItem**

Compiler Version **v0.8.13+commit.abaa5c0e**

Optimization Enabled: **No** with **200** runs

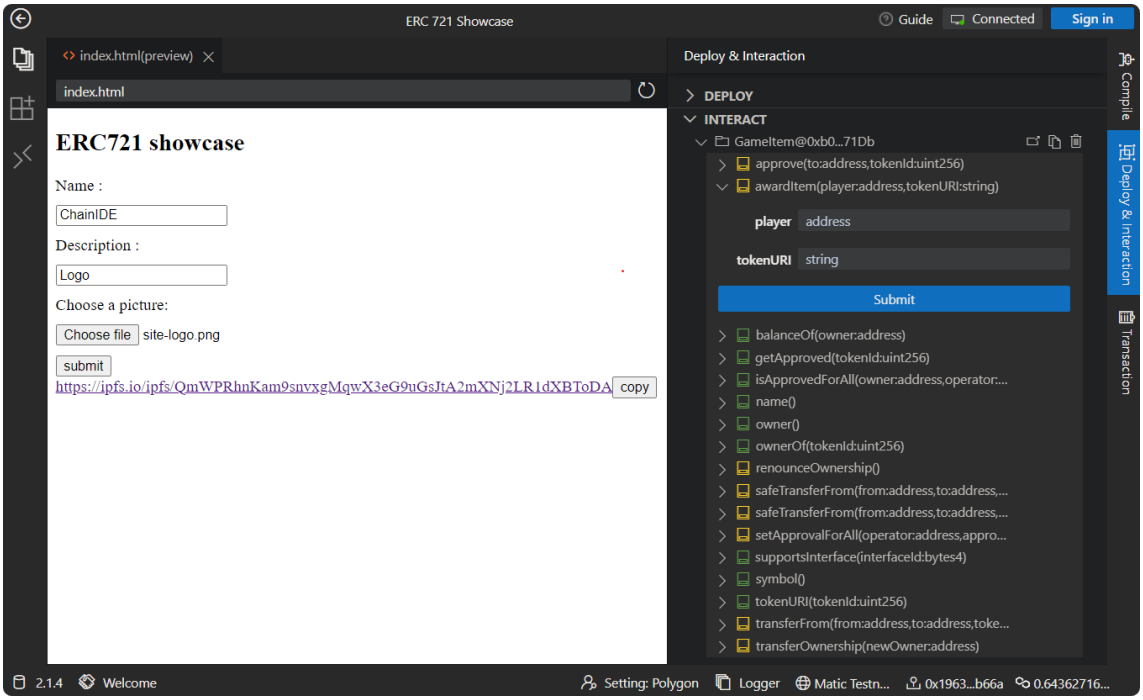
Other Settings: **default** evmVersion, **MIT** license

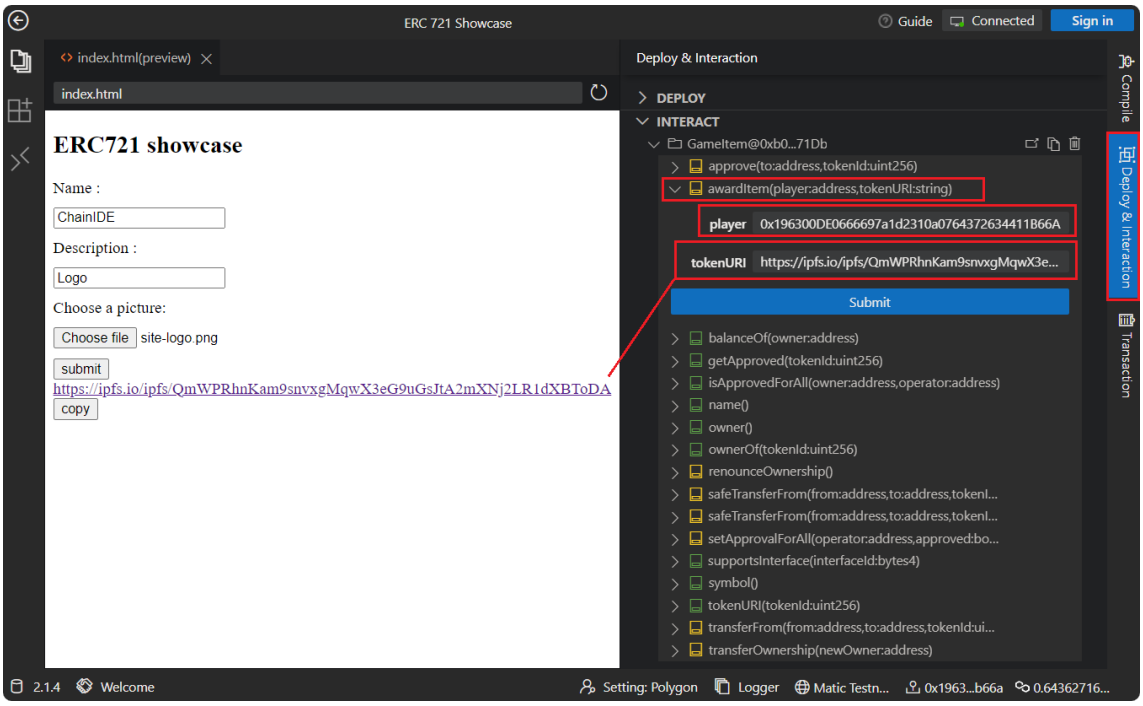
Contract Source Code (Solidity)

Outline | More Options

Minting NFT

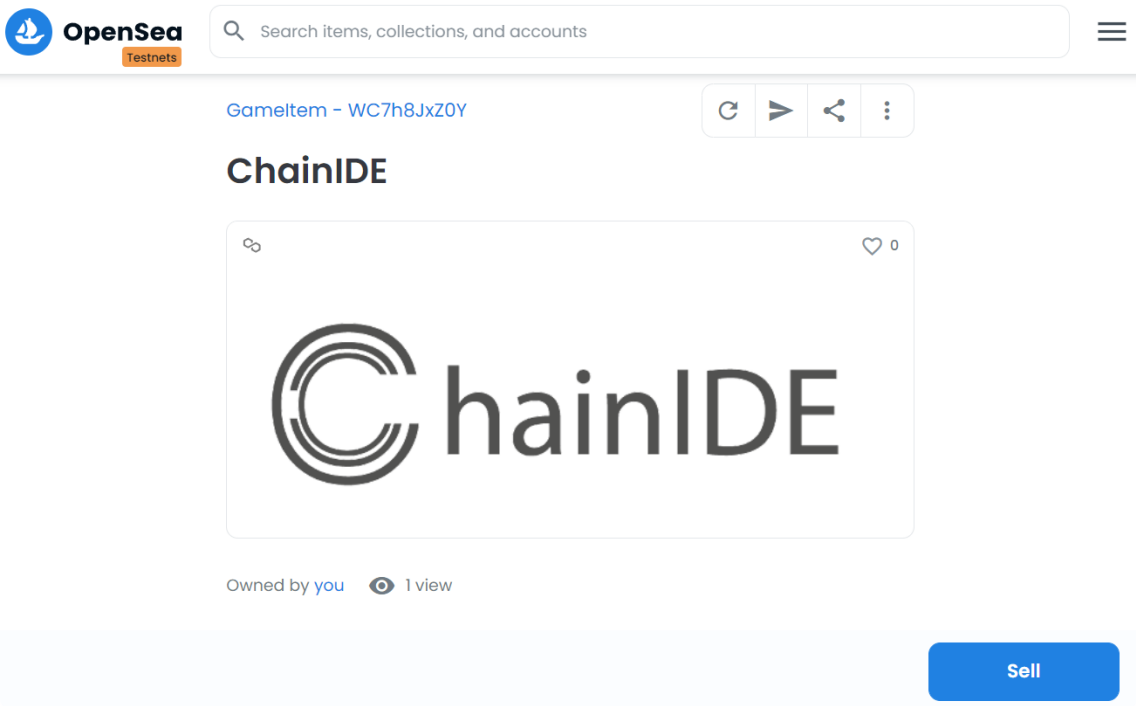
To mint an NFT, you need to use the **Award Item** function, the wallet address of someone to whom you want to award an NFT, and the link of the photo uploaded to IPFS to be pasted in the token URL input field.





After successful minting, you can check the minted NFT on the OpenSea NFT marketplace. Visit OpenSea Testnet, connect your MetaMask wallet and make sure the selected network is Polygon Mumbai Testnet, and you'll be able to see and trade the minted NFT on the OpenSea NFT marketplace.

Congratulations! You have successfully minted an NFT on Polygon using ChainIDE.



---

id: chainstack title: How Deploy a Smart Contract Using Chainstack and Foundry sidebar_label: How Deploy a Smart Contract Using Chainstack description: Use Chainstack and Foundry to develop a Smart Contract on Polygon keywords: - docs - matic - polygon - build - deploy smart contract - chainstack - foundry image: https://wiki.polygon.technology/img/polygon-logo.png

:::caution Content disclaimer

Please view the third-party content disclaimer here.

:::

Overview

This section guides you through deploying a Hello World contract using Chainstack and Foundry on the Polygon Mumbai testnet.

Chainstack provides infrastructure for Ethereum-based applications and other blockchains. They maintain nodes and guarantee their connection to the network and also offer an interface to interact with mainnet and testnets.

Foundry is a fast toolkit for Ethereum application development written in Rust. It provides testing, interaction with EVM smart contracts, sending transactions, and blockchain data retrieval.

:::tip

If you have any questions, reach out in the Chainstack Discord server.

:::

What you will learn

Create a Hello World contract, using Chainstack to deploy a Polygon node and Foundry to deploy the contract.

What you will do

1. Deploy a Polygon node using Chainstack
2. Set up Foundry
3. Create the smart contract
4. Deploy the smart contract.

Deploy a Polygon Mumbai Node

You need a node to deploy a smart contract to the blockchain network. Follow the steps below to get your node up and running:

Step 1 → Sign up with Chainstack

img

Step 2 → Follow the instructions on how to deploy a Mumbai node

img

Step 3 → Get the deployed node's HTTPS endpoint

Install Foundry

Foundry is a development toolkit to work with smart contracts. To begin working with it, you need to install the Rust coding language first.

1. Install Rust.
2. Install Foundry.

Initialize with Foundry

To create a boilerplate project, navigate to your working directory and run:

```
forge init PROJECT_NAME
// PROJECT_NAME - name of project
```

Fund Your Account

You will need a wallet account to deploy the smart contract. You can use Metamask for that. You also need to pay gas on the network to deploy the contract. Just copy your wallet address and get Mumbai MATIC token through the faucet.

Create the Hello World contract

In the initialized Foundry project in `src/`, create `HelloWorld.sol`:

```
// SPDX-License-Identifier: None

// Specifies the version of Solidity, using semantic versioning.
// Learn more: https://solidity.readthedocs.io/en/v0.5.10/layout-of-source-files.html#pragma
pragma solidity >=0.8.9;

// Defines a contract named `HelloWorld`.
// A contract is a collection of functions and data (its state). Once deployed, a contract resides at a specific address on the
Ethereum blockchain. Learn more: https://solidity.readthedocs.io/en/v0.5.10/structure-of-a-contract.html
contract HelloWorld {

   //Emitted when update function is called
   //Smart contract events are a way for your contract to communicate that something happened on the blockchain to your app front-
end, which can be 'listening' for certain events and take action when they happen.
   event UpdatedMessages(string oldStr, string newStr);

   // Declares a state variable `message` of type `string`.
   // State variables are variables whose values are permanently stored in contract storage. The keyword `public` makes variables
accessible from outside a contract and creates a function that other contracts or clients can call to access the value.
   string public message;

   // Similar to many class-based object-oriented languages, a constructor is a special function that is only executed upon
contract creation.
   // Constructors are used to initialize the contract's data. Learn
more:https://solidity.readthedocs.io/en/v0.5.10/contracts.html#constructors
   constructor(string memory initMessage) {

      // Accepts a string argument `initMessage` and sets the value into the contract's `message` storage variable).
      message = initMessage;
   }

   // A public function that accepts a string argument and updates the `message` storage variable.
   function update(string memory newMessage) public {
      string memory oldMsg = message;
      message = newMessage;
      emit UpdatedMessages(oldMsg, newMessage);
   }
}
```

Deploy the Contract

At this point, you are ready to deploy your contract:

- You have your own node on the Polygon Mumbai network through which you will deploy the contract.
- You have Foundry that you will use to deploy the contract.

- You have a funded account that will deploy the contract.

To deploy the contract, run:

```
forge create HelloWorld --constructor-args "Hello" --contracts CONTRACT_PATH --private-key PRIVATE_KEY --rpc-url HTTPS_ENDPOINT
```

Here,

- CONTRACT_PATH — path to your `HelloWorld.sol` file.
- PRIVATE_KEY — the private key from your account.
- HTTPS_ENDPOINT — your node's endpoint.

Example:

```
forge create HelloWorld --constructor-args "Hello" --contracts /root/foundry/src/HelloWorld.sol --private-key
d8936f6eae35c73a14ea7c1aabb8d068e16889a7f516c8abc482ba4e1489f4cd --rpc-url https://nd-123-456-
789.p2pify.com/3c6e0b8a9c15224a8228b9a98ca1531d
```

:::tip

You can always check the contract's deployment on Mumbai Polygonscan using the newly-generated hash from the last step.

:::

Test the Contract

There is a `forge test` command in case you need to check whether the contract is working fine. Foundry provides many options (flags) for more specific tests. Learn more about writing tests, advanced testing and other features at Foundry's documentation.

Congratulations! You have deployed your Hello World smart contract on Polygon.

See also Chainstack docs for more Polygon-related tutorials and tools.

id: getblock title: How to Deploy a Smart Contract using GetBlock sidebar_label: How to Deploy a Smart Contract using GetBlock description: Guide to deploy smart contracts using GetBlock keywords: - docs - matic - polygon - getblock - create smart contract - deploy on polygon - verify your smart contract image: https://wiki.polygon.technology/img/polygon-logo.png

:::caution Content disclaimer

Please view the third-party content disclaimer here.

:::

The Polygon API endpoint by GetBlock is a useful tool for early-stage teams that want to take advantage of Polygon's benefits. In this tutorial, we will walk you through the steps to create and deploy an ERC-20 smart contract on the Polygon Mumbai testnet using Metamask, Remix, and GetBlock.

You will learn how to use GetBlock to create and deploy a smart contract on the Mumbai Testnet using GetBlock endpoints.

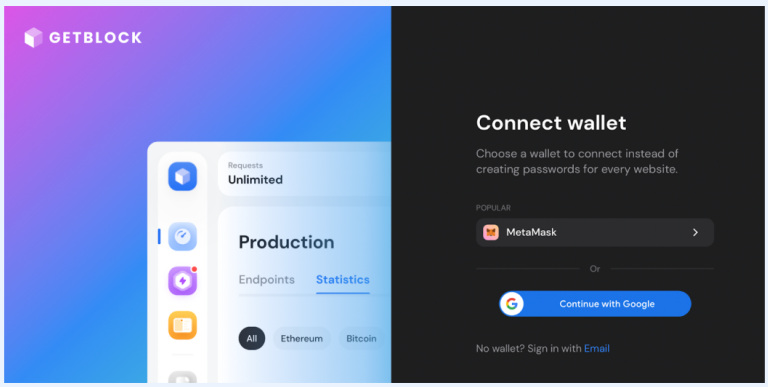
What you will do

In this tutorial, you will:

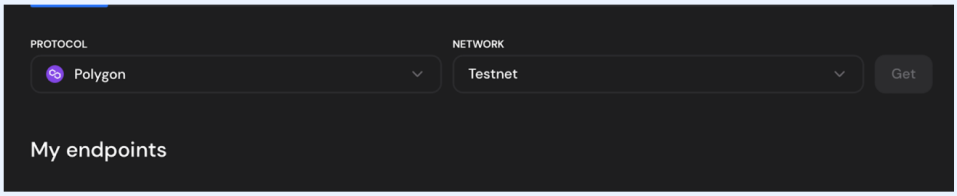
1. Get started on GetBlock and access test nodes.
2. Create a wallet address with Metamask.
3. Add test tokens to your wallet.
4. Compile and deploy a smart contract using Remix
5. Check the status of your smart contract.

Preparation

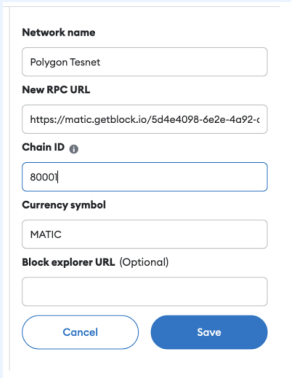
1. The first step is to sign up with Getblock via wallet or email via



2. Create an endpoint for Polygon testnet (Mumbai).



3. Then enter Metamask and choose to connect a network manually, connect your Metamask to the Mumbai Testnet adding your GetBlock endpoint in the **RPC URL** line



If you don't have any tokens in your account, you can request some from the Mumbai faucet through the faucet website.

Smart Contract Development

1. Visit the official Remix website. Remix is a Web-based IDE for Ethereum smart contract development in Solidity.
2. Click on the "Create a new file" button located on the left side of the page.
3. Select “Storage.sol” from Remix as an example of a contract.

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.7.0 <0.9.0;

/**
 * @title Storage
 * @dev Store & retrieve value in a variable
 * @custom:dev-run-script ./scripts/deploy_with_ethers.ts
 */
contract Storage {

    uint256 number;

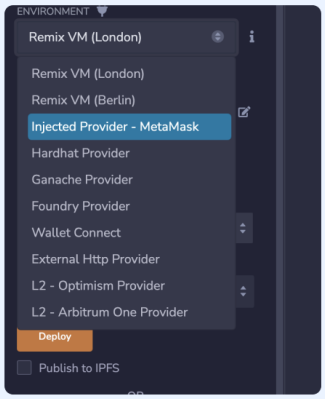
    /**
     * @dev Store value in variable
     * @param num value to store
     */
    function store(uint256 num) public {
        number = num;
    }

    /**
     * @dev Return value
     * @return value of 'number'
     */
    function retrieve() public view returns (uint256){
        return number;
    }
}
```

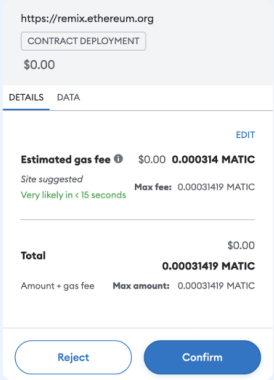
4. Deploy the smart contract by clicking on the “Deploy & Run Transactions” tab.
5. Click on the “Deploy & Run Transactions” tab in Remix, located on the left-hand side of the screen.
6. In the “Contract” dropdown menu, select the name of the smart contract you want to deploy and click on the “Deploy” button to deploy the smart contract. A Metamask window will pop up asking you to confirm the transaction. Click on the “Confirm” button to proceed.Wait for the transaction to be confirmed on the Polygon network.

> *You can track the status of the transaction in the Metamask window or by checking the transaction hash on the Polygon explorer.*

Once the transaction has been confirmed, you should see a green checkmark and a message indicating that the contract has been deployed successfully.

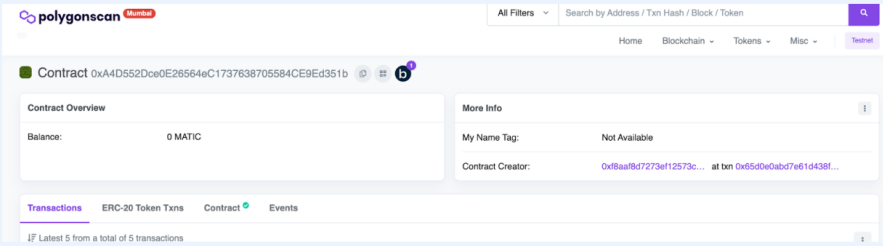


7. Locate the storage variable you want to test.
8. Choose "Injected provider" as the provider and select "Metamask" as the wallet.



Check that your smart contract has been created on the network by looking at the Remix output panel.

9. Click on the "Polygon (Matic) Mainnet" dropdown menu located at the top right corner of the page and select "Mumbai Testnet" from the list of available networks. Then enter the address of your smart contract in the search bar located at the top of the page.
10. Click on the search button or hit enter.



Smart contract URL: https://mumbai.polygonscan.com/address/0xa4d552dce0e26564ec1737638705584ce9ed351b

If your smart contract has been successfully deployed, you should see the contract details including its address, balance, and transaction history. Congratulations, you have successfully deployed a smart contract on the Mumbai Testnet!

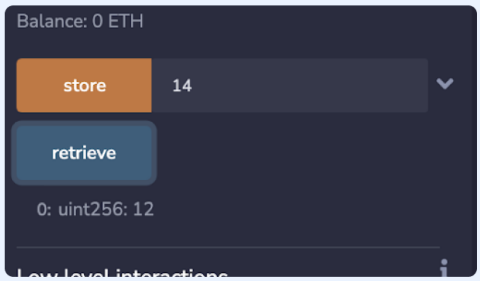
Testing and Verification

Once you have deployed your smart contract on the Polygon test network using GetBlock, it is important to test and verify its functionality to ensure that it works as intended. Here are some steps you can take for testing and verification:

1. Open the smart contract in Remix and locate the storage variable you want to test. In this example, we will use a variable called "storage" that initially has a value of 12."

2. Change the value of the variable to 14 by entering "14" in the "Store" field and saving the value on the network after confirming the operation in Metamask.

3. Click on the "Transact" button to confirm the transaction in Metamask.



4. Wait for the transaction to be added to a network block. Once the transaction is confirmed, navigate to the "Read Contract" section under "Deployed Contracts".

5. Find the "storage" variable and click on the "Read" button. The new value of the variable should now be displayed as "14".

By following these steps, you can ensure that your smart contract works as intended and is secure and reliable for your users.

Congratulations, you have successfully deployed a smart contract on the Mumbai Testnet!

If you have any additional questions or would like to share your experience, feel free to join our community of Web3 developers who are always ready to chat.

id: quicknode title: How to Deploy a Smart Contract Using QuickNode sidebar_label: How to Deploy a Smart Contract Using QuickNode description: Deploy Smart Contracts on Polygon using Brownie and Quicknode. keywords: - docs - matic - quicknode - polygon - python - web3.py - smart contract - brownie - deploy image: https://wiki.polygon.technology/img/polygon-logo.png

:::caution Content disclaimer

Please view the third-party content disclaimer here.

:::

Overview

Python is one of the most versatile programming languages; from researchers running their test models to developers using it in heavy production environments, it has use cases in every possible technical field.

In this tutorial, you will learn how to use Brownie framework to write and deploy a smart contract by leveraging QuickNode testnet nodes for Polygon.

:::tip

To contact the Quicknode team, send them a message or tag them on Twitter @QuickNode.

:::

Prerequisites

- Python3 installed
- A Polygon node
- Code editor
- Command line interface

What you will do

1. Set up Brownie
2. Get access to Quicknode test nodes
3. Compile and Deploy a smart contract
4. Check the deployed contract data

What is Brownie?

Smart contract development is majorly dominated by JavaScript-based libraries like web3.js, ethers.js, Truffle, and Hardhat. Python is a versatile, highly used language and can also be used for smart contracts / Web3 development; web3.py is a compelling Python library that fulfills Web3 needs. Brownie framework is built on top of `web3.py`.

Brownie is a Python-based framework to develop and test smart contracts. Brownie has support for both Solidity and Vyper contracts, and it even provides contract testing via pytest.

To demonstrate the process of writing and deploying a smart contract with Brownie, we will use Brownie-mixes which are template projects. Specifically, we will use a token mix, which is a template of the ERC-20 implementation.

Install dependencies

Brownie is built on top of python3, so we need it installed to work with Brownie. Let us check if we have python3 installed on our system. To do so, type the following in your command line tool:

```
python3 -V
```

This should return the version of python3 installed. If not installed, download and install it from the official Python website.

Let us make a project directory before installing Brownie, and make that project directory our current working directory:

```
mkdir brownieDemo
cd brownieDemo
```

Now that you have installed python3 on your system, let us install Brownie using pip, Python's package manager. Pip is similar to what npm is for JavaScript. Type the following in your command line:

```
pip3 install eth-brownie
```

:::tip

If the install fails, you can use the following command instead: `sudo pip3 install eth-brownie`

:::

To check if Brownie was installed correctly, type `brownie` in your command line, and it should give the following output:

img

To get the token mix, simply type the following in your command line:

```
brownie bake token
```

This will create a new directory `token/` in our `brownieDemo` directory.

File structure

First of all, navigate to the `token` directory:

```
cd token
```

Now, open the `token` directory in your text editor. Under the `contracts/` folder you will find `Token.sol`, which is our main contract. You can write your own contracts or modify `Token.sol` file.

Under the `scripts/` folder, you will find `token.py` Python script. This script will be used to deploy the contract, and modifications are needed based on contracts.

img

The contract is an ERC-20 contract. You can learn more about the ERC-20 standards and contracts in this guide on ERC-20 tokens.

Booting your Polygon node

QuickNode has a global network of Polygon Mainnet and Mumbai testnet nodes. They also run a free public Polygon RPC but if you get rate limited, you can sign up for a free trial node from QuickNode.

img

Copy the **HTTP URL**, which will be useful later in the tutorial.

Network and Account setup

We need to set up our QuickNode endpoint with Brownie. To do so, type the following in your command line:

```
brownie networks add Ethereum matic_mumbai host=YOUR_QUICKNODE_URL chainid=3
```

Replace `YOUR_QUICKNODE_URL` with the **Mumbai Testnet HTTP URL** that we just received while booting our Polygon node.

In the above command, `Ethereum` is the name of the environment, and `matic_mumbai` is the custom name of the network; you can give any name to your custom network.

The next thing we need to do here is to create a new wallet using Brownie, to do so type the following in your command line:

```
brownie accounts generate testac
```

You will be asked to set up a password for your account! After completing the steps, this will generate an account along with a mnemonic phrase, save it offline. The name `testac` is the name for our account (You can choose any name that you like).

img

:::note

Mnemonic phrases can be used to recover an account or import the account to other non-custodial wallets. The account you see in the image above was just created for this guide.

:::

Copy the account address so that we can get some test MATIC, which will be required to deploy our contract.

Getting Testnet MATIC

We will need some test MATIC tokens to pay for gas fees to deploy our smart contract.

Copy the address of your account which we generated in this tutorial, paste it into the address field of Polygon faucet, and click on **Submit**. The faucet will send you 0.2 test MATIC.

img

Deploying your Smart Contract

Before deploying the contract, you need to compile it using:

```
brownie compile
```

img

Now open the `scripts/token.py` in your text editor, and make the following changes:

```
#!/usr/bin/python3
from brownie import Token, accounts

def main():
    acct = accounts.load('testac')
    return Token.deploy("Test Token", "TST", 18, 1e21, {'from': acct})
```

:::info Explanation

Using the above code, we have imported `testac` account which we created earlier, and stored it in `acct` variable. Also, in the next line, we have edited `'from':` part to receive data from `acct` variable.

:::

Finally, we will deploy our smart contract:

```
brownie run token.py --network matic_mumbai
```

`matic_mumbai` is the name of the custom network which we created earlier. The prompt will ask you for the **password** that we set earlier while making the account.

After running the above command, you must get the transaction hash, and Brownie will wait for the transaction to get confirmed. Once the transaction is confirmed, it will return the address at which our contract is deployed on the Polygon Mumbai testnet.

img

You can check out the deployed contract by copy-pasting the contract address at Polygonscan Mumbai.

img

Testing the Contract

Brownie also offers the option of testing smart contracts functionalities. It uses the `pytest` framework to easily generate unit tests. You can find more information about writing tests on Bronwnie on their documentation.

This is how contracts are deployed on Polygon using Brownie and QuickNode.

QuickNode, just like Polygon, has always had an education-first approach providing developer guides, docs, tutorial videos and a community of Web3 developers who are eager to help each other.

id: thirdweb title: How to Deploy a Smart Contract Using thirdweb sidebar_label: How to Deploy a Smart Contract Using thirdweb description: "Lean how to deploy smart contracts on Polygon PoS mainnet using thirdweb." keywords: - docs - matic - polygon - thirdweb - create smart contract - deploy on polygon image: https://wiki.polygon.technology/img/polygon-logo.png

:::caution Content disclaimer

Please view the third-party content disclaimer here.

:::

thirdweb is a complete web3 development framework that provides everything you need to connect your apps and games to decentralized networks.

Create a contract using thirdweb

To create a new smart contract using thirdweb CLI, follow the step below.

1. In your CLI, run the following command to create a contract:

```
npx thirdweb create contract
```

2. Enter your preferred values for the command line prompts: i. Give your project a name. ii. Choose your preferred framework: Hardhat or Foundry. iii. Name your smart contract. iv. Choose the type of base contract: Empty, ERC20, ERC721, or ERC1155. v. Add any desired extensions.

3. Once created, navigate to your project's root directory and open your project in your preferred code editor.

4. If you open the `contracts` folder, you will find your smart contract; the contract you see is your smart contract written in Solidity.

   The following code is for an ERC721Base contract without specified extensions. It implements all of the logic inside the `[ERC721Base.sol](https://github.com/thirdweb-dev/contracts/blob/main/contracts/base/ERC721Base.sol)` contract, which implements the `[ERC721A](https://github.com/thirdweb-dev/contracts/blob/main/contracts/eip/ERC721A.sol)` standard. You can view the full code for this sample contract on thirdweb's GitHub.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "@thirdweb-dev/contracts/base/ERC721Base.sol";

contract Contract is ERC721Base {
    constructor(
        string memory _name,
        string memory _symbol,
        address _royaltyRecipient,
        uint128 _royaltyBps
    ) ERC721Base(_name, _symbol, _royaltyRecipient, _royaltyBps) {}
}
```

   This contract inherits the functionality of ERC721Base through the following steps:

   - Imports the ERC721Base contract.
   - Inherits the contract by declaring that our contract is an ERC721Base contract.
   - Implements the required methods, including the constructor.

5. After modifying your contract with your desired custom logic, you may deploy it to Polygon PoS mainnet using Deploy.

Alternatively, you can deploy a prebuilt contract an NFTs, token, or marketplace directly from the thirdweb Explore page. To do so, follow the steps below.

1. Navigate to the thirdweb Explore page:

   img

2. Choose the type of contract you want to deploy from the available options.

3. Follow the on-screen prompts to configure and deploy your contract.

> *For more information on different contracts available on Explore, check out thirdweb's documentation.*

Deploy a contract using thirdweb

Deploy allows you to deploy a smart contract to any EVM-compatible network without configuring RPC URLs, exposing your private keys, writing scripts, and other additional setup such as verifying your contract.

1. To deploy your smart contract using deploy, navigate to the root directory of your project and execute the following command:

```
npx thirdweb deploy
```

   Executing this command will trigger the following actions:

- Compiles all the contracts in the current directory.
- Provides the option to select which contract(s) you wish to deploy.
- Uploads your compiled smart contract code (in the form of an Application Binary Interface, or ABI) to IPFS.

2. When deployment is complete, a dashboard interface will open to fill out the remaining parameters.
   - `_name` : contract name
   - `_symbol` : symbol or "ticker"
   - `_royaltyRecipient` : wallet address to receive royalties from secondary sales
   - `_royaltyBps` : basis points (bps) that will be given to the royalty recipient for each secondary sale, e.g. 500 = 5%
3. Select Polygon as the network.
4. Once your smart contract is deployed, you can manage additional settings and functionalities through its dashboard. For example, you can upload NFTs, configure permissions and access control, add new features.

For additional information on Deploy, please reference thirdweb's documentation.

Verify the Contract

Contracts deployed to Polygon are automatically verified by thirdweb. You can check the status on the "Sources" view in the thirdweb dashboard.

Need help?

If you have any questions or encounter any issues during the process, please reach out to the official thirdweb support.

id: bor title: Bor Architecture description: The Bor role in the Polygon architecture keywords: - docs - matic - Bor Architecture - polygon image: https://matic.network/banners/matic-network-16x9.png

import useBaseUrl from '@docusaurus/useBaseUrl';

Bor Architecture

Polygon PoS mainnet uses dual-consensus architecture on the Polygon Network to optimise for speed and decentralisation.

Architecture

<img src={useBaseUrl("img/Bor/matic_structure.png")}/>

A blockchain is a set of network clients interacting and working together. The client is a piece of software capable of establishing a p2p communication channel with other clients, signing and broadcasting transactions, deploying and interacting with smart contracts, etc. The client is often referred to as a node.

For Polygon, the node is designed with a two layer implementation Heimdall (Validator Layer) and Bor(Block Producer Layer).

1. Heimdall
   - Proof-of-Stake verification
   - Checkpointing blocks on Ethereum main chain
   - Validator and Rewards Management
   - Ensuring Sync with Ethereum main chain
   - Decentralised Bridge
2. Bor
   - Polygon Chain
   - EVM Compatible VM
   - Proposers and Producer set selection
   - SystemCall
   - Fee Model

Heimdall (Validator layer)

Heimdall (the All-Protector) is the purveyor of all that happens in the Polygon Proof-of-Stake system – good or bad.

Heimdall is our Proof-of-Stake Verifier layer, which is responsible for checkpointing a representation of the blocks to the main chain in our architecture. We have implemented this by building on top of the Tendermint consensus engine with changes to the signature scheme and various data structures.

For more information, please read https://blog.matic.network/heimdall-and-bor-matic-validator-and-block-production-layers/.

Bor (Block Producer layer)

The Bor node implementation is basically the EVM-compatible blockchain operator. Currently, it is a basic Geth implementation with custom changes done to the consensus algorithm. However, this will be built from the ground up to make it lightweight and focused.

Bor is our Block producer layer, which in sync with Heimdall selects the producers and verifiers for each span and sprint.

Polygon Chain

This chain is a separate blockchain that is attached to Ethereum using a two-way peg. The two-way peg enables interchangeability of assets between the Ethereum and Polygon.

EVM Compatible VM

The Ethereum Virtual Machine (EVM) is a powerful, sandboxed virtual stack embedded within each full Polygon node, responsible for executing contract bytecode. Contracts are typically written in higher level languages, like Solidity, then compiled to EVM bytecode.

Proposers and Producers Selection

Block Producers for the Bor layer are a committee selected from the Validator pool on the basis of their stake, which happens at regular intervals and is shuffled periodically. These intervals are decided by the Validator's governance with regards to dynasty and network.

Ratio of Stake/Staking power specifies the probability to be selected as a member of the block producer committee.

<img src={useBaseUrl("img/Bor/bor-span.png")} />

Selection Process

- Let's suppose we have 3 validators in pool, and they are Alice, Bill and Clara.
- Alice staked 100 Matic tokens whereas Bill and Clara staked 40 Matic tokens.
- Validators are given slots according to the stake, as Alice has 100 Matic tokens staked, she will get slots proportionally. Alice will get 5 slots in total. Similarly, Bill and Clara get 2 slots in total.
- All the validators are given these slots [A, A, A, A, A, B, B, C, C]
- Using historical Ethereum block data as seed, we shuffle this array.
- After shuffling the slots using the seed, say we get this array [A, B, A, A, C, B, A, A, C]
- Now depending on Producer count*(maintained by validator's governance)*, we pop validators from the top. For e.g. if we want to select 5 producers we get the producer set as [A, B, A, A, C]
- Hence the producer set for the next span is defined as [A: 3, B:1, C:1].
- Using this validator set and tendermint's proposer selection algorithm we choose a producer for every sprint on BOR.

SystemCall Interface

System call is an internal operator address which is under EVM. This helps to maintain the state for Block Producers for every sprint. A System Call is triggered towards the end of a sprint and a request is made for the new list of Block Producers. Once the state is updated, changes are received after block generation on Bor to all the Validators.

Functions

proposeState

- Call is only allowed to validators.
- Inspect `stateId` if it is already proposed or committed.
- Propose the `stateId` and update the flag to `true`.

commitState

- Call is only allowed to System.
- Inspect `stateId` if it is already proposed or committed.
- Notify `StateReceiver` Contract with new `stateId`.
- Update the `state` flag to `true`, And `remove` the `proposedState`.

proposeSpan

- Call is only allowed to validators.
- Check if the Span proposal is `pending`.
- Update the Span Proposal to `true`

proposeCommit

- Call is only allowed to System.
- Set `initial validators` if current span is zero.
- Check Conditions for `spanId` and `time_period` of Sprint and Span.
- Update the new `span` and `time_period`.
- Set `validators` and `blockProducers` for the `sprint`.
- Update the flag for `spanProposal` to `true`.

Bor Fee Model

For normal transaction, fees in Matic token gets collected and distributed to block producers, similar to Ethereum transactions.

Like other blockchains, Polygon has a native token called Matic(MATIC). MATIC is an ERC20 token used primarily for paying gas(transaction fees) on Polygon and staking.

:::info

An important thing to note is that on the Polygon chain, the MATIC tokens works as an ERC20 token, but also as the native token - both at the same time. Therefore, this means that a user can pay gas with MATIC as well as send MATIC to other accounts.

:::

For genesis-contracts, `gasPrice` and `gasLimit` works same as Ethereum, but during the execution it won't deduct the fees from sender's account.

Genesis transactions from current validators are executed with `gasPrice = 0`.

Also, validators have to send following types of transaction like State proposals like deposits & Span proposals on Bor.

Technical Insight

Genesis Contracts

BorValidatorSet(0x1000) ⇒ This contract manages validator set for each span and sprint.

BorStateReceiver(0x1001) ⇒ This Contract manages the transfer of arbitrary contract data from Ethereum contracts to Polygon contracts

MaticChildERC20(0x1010) ⇒ Child Contract for Main Chain tokens which allows to move assets from Ethereum to Polygon.

Bor.go

Bor Protocol

Glossary

- StartEpoch - Checkpoint number post which a validator is activated and will participate in the consensus.
- EndEpoch - Checkpoint number post which a validator is considered deactivated and won't participate in the consensus.
- Sprint - Sprint is a continuous set of blocks created by a single validator.
- Span - Span is a big set of blocks with a fixed validator set but consisting of various sprints. For eg for a span of length 1600 blocks it will consist of 100 sprints of 16 blocks.
- Dynasty: Time between the end of last auction and start time of next auction.

Resources

- Bor
- EVM
- How EVM Works?
- Tendermint Proposer Selection

---

id: consensus title: Bor Consensus description: Bor mechanism for fetching new producers keywords: - docs - matic - Bor Consensus - polygon image: https://matic.network/banners/matic-network-16x9.png

import useBaseUrl from '@docusaurus/useBaseUrl';

Bor Consensus

Bor consensus is inspired by Clique consensus: https://eips.ethereum.org/EIPS/eip-225. Clique works with multiple pre-defined producers. All producers vote on new producers using Clique APIs. They take turns creating blocks.

Bor fetches new producers through span and sprint management mechanism.

Validators

Polygon is a Proof-of-stake system. Anyone can stake their Matic token on Ethereum smart-contract, "staking contract", and become a validator for the system.

```
function stake(
    uint256 amount,
    uint256 heimdallFee,
    address signer,
    bool acceptDelegation
) external;
```

Once validators are active on Heimdall they get selected as producers through `bor` module.

Check Bor overview to understand span management more in details: Bor Overview

Span

A logically defined set of blocks for which a set of validators is chosen from among all the available validators. Heimdall provides span details through span-details APIs.

```
// HeimdallSpan represents span from heimdall APIs
type HeimdallSpan struct {
    Span
    ValidatorSet      ValidatorSet `json:"validator_set" yaml:"validator_set"`
    SelectedProducers []Validator  `json:"selected_producers" yaml:"selected_producers"`
    ChainID           string       `json:"bor_chain_id" yaml:"bor_chain_id"`
}

// Span represents a current bor span
type Span struct {
    ID         uint64 `json:"span_id" yaml:"span_id"`
    StartBlock uint64 `json:"start_block" yaml:"start_block"`
    EndBlock   uint64 `json:"end_block" yaml:"end_block"`
}

// Validator represents a volatile state for each Validator
type Validator struct {
```

```
    ID               uint64         `json:"ID"`
    Address          common.Address `json:"signer"`
    VotingPower      int64          `json:"power"`
    ProposerPriority int64          `json:"accum"`
}
```

Geth (In this case, Bor) uses block `snapshot` to store state data for each block, including consensus related data.

Each validator in span contains voting power. Based on their power, they get selected as block producers. Higher power, a higher probability of becoming block producers. Bor uses Tendermint's algorithm for the same. Source: https://github.com/maticnetwork/bor/blob/master/consensus/bor/valset/validator_set.go

Sprint

A set of blocks within a span for which only a single block producer is chosen to produce blocks. The sprint size is a factor of span size. Bor uses `validatorSet` to get current proposer/producer for current sprint.

```
currentProposerForSprint := snap.ValidatorSet().Proposer
```

Apart from the current proposer, Bor selects back-up producers.

Authorizing a block

The producers in Bor also called signers, since to authorize a block for the network, the producer needs to sign the block's hash containing **everything except the signature itself**. This means that the hash contains every field of the header, and also the `extraData` with the exception of the 65-byte signature suffix.

This hash is signed using the standard `secp256k1` curve, and the resulting 65-byte signature is embedded into the `extraData` as the trailing 65-byte suffix.

Each signed block is assigned to a difficulty that puts weight on Block. In-turn signing weighs more (`DIFF_INTURN`) than out of turn one (`DIFF_NOTURN`).

Authorization strategies

As long as producers conform to the above specs, they can authorize and distribute blocks as they see fit. The following suggested strategy will, however, reduce network traffic and small forks, so it's a suggested feature:

- If a producer is allowed to sign a block (is on the authorized list)
  - Calculate the optimal signing time of the next block (parent + `Period`)
  - If the producer is in-turn, wait for the exact time to arrive, sign and broadcast immediately
  - If the producer is out-of-turn, delay signing by `wiggle`

This small strategy will ensure that the in-turn producer (who's block weighs more) has a slight advantage to sign and propagate versus the out-of-turn signers. Also, the scheme allows a bit of scale with an increase of the number of producers.

Out-of-turn signing

Bor chooses multiple block producers as a backup when in-turn producer doesn't produce a block. This could happen for a variety of reasons like:

- Block producer node is down
- The block producer is trying to withhold the block
- The block producer is not producing a block intentionally.

When the above happens, Bor's backup mechanism kicks in.

At any point of time, the validators set is stored as an array sorted on the basis of their signer address. Assume, that the validator set is ordered as A, B, C, D and that it is C's turn to produce a block. If C doesn't produce a block within a sufficient amount of time, D becomes in turn to produce one. If D doesn't then A and then B.

However, since there will be some time before C produces and propagates a block, the backup validators will wait a certain amount of time before starting to produce a block. This time delay is called wiggle.

Wiggle

Wiggle is the time that a producer should wait before starting to produce a block.

- Say the last block (n-1) was produced at time `t` .
- We enforce a minimum time delay between the current and next block by a variable parameter `Period` .
- In ideal conditions, C will wait for `Period` and then produce and propagate the block. Since block times in Polygon are being designed to be quite low (2-4s), the propagation delay is also assumed to be the same value as `Period` .
- So if D doesn't see a new block in time `2 * Period` , D immediately starts producing a block. Specifically, D's wiggle time is defined as `2 * Period * (pos(d) - pos(c))` where `pos(d) = 3` and `pos(c) = 2` in the validator set. Assuming, `Period = 1` , wiggle for D is 2s.
- Now if D also doesn't produce a block, A will start producing one when the wiggle time of `2 * Period * (pos(a) + len(validatorSet) - pos(c)) = 4s` has elapsed.
- Simmilary, wiggle for C is `6s`

Resolving forks

While the above mechanism adds to the robustness of chain to a certain extent, it introduces the possibility of forks. It could actually be possible that C produced a block, but there was a larger than expected delay in propagation and hence D also produced a block, so that leads to at least 2 forks.

The resolution is simple - choose the chain with higher difficulty. But then the question is how do we define difficulty of a block in our setup?

Difficulty

- The difficulty for a block that is produced by an in-turn signer (say c) is defined to be the highest = `len(validatorSet)` .
- Since D is the producer who is next in line; if and when the situation arises that D is producing the block; the difficulty for the block will be defined just like in wiggle as `len(validatorSet) - (pos(d) - pos(c))` which is `len(validatorSet) - 1`
- Difficulty for block being produced by A while acting as a backup becomes `len(validatorSet) - (pos(a) + len(validatorSet) - pos(c))` which is `2`

Now having defined the difficulty of each block, the difficulty of a fork is simply the sum of the difficulties of the blocks in that fork. In the case when a fork has to be chosen, the one with higher difficulty is chosen, since that is a reflection of the fact that blocks were produced by in-turn block producers. This is simply to provide some sense of finality to the user on Bor.

View Change

After each span, Bor changes view. It means that it fetches new producers for the next span.

Commit span

When the current span is about to end (specifically at the end of the second-last sprint in the span), Bor pulls a new span from Heimdall. This is a simple HTTP call to the Heimdall node. Once this data is fetched, a `commitSpan` call is made to the BorValidatorSet genesis contract through System call.

Bor also sets producers bytes into the header of the block. This is necessary while fast-syncing Bor. During fast-sync, Bor syncs headers in bulk and validates if blocks are created by authorized producers.

At the start of each Sprint, Bor fetches header bytes from the previous header for next producers and starts creating blocks based on `ValidatorSet` algorithm.

Here is how header looks like for a block:

```
header.Extra = header.Vanity + header.ProducerBytes /* optional */ + header.Seal
```

<img src={useBaseUrl("img/Bor/header-bytes.svg")} />

State sync from Ethereum Chain

Bor provides a mechanism where some specific events on the main Ethereum chain are relayed to Bor.

1. Any contract on Ethereum may call syncState in `StateSender.sol` . This call emits `StateSynced` event: https://github.com/maticnetwork/contracts/blob/develop/contracts/root/stateSyncer/StateSender.sol#L38

```
event StateSynced(uint256 indexed id, address indexed contractAddress, bytes data)
```

2. Heimdall listens to these events and calls `function proposeState(uint256 stateId)` in `StateReceiver.sol` - thus acting as a store for the pending state change ids. Note that the `proposeState` transaction will be processed even with a 0 gas fee as long as it is made by one of the validators in the current validator set: https://github.com/maticnetwork/genesis-contracts/blob/master/contracts/StateReceiver.sol#L24
3. At the start of every sprint, Bor pulls the details about the pending state changes using the states from Heimdall and commits them to the Bor state using a system call. See `commitState` here: https://github.com/maticnetwork/genesis-contracts/blob/f85d0409d2a99dff53617ad5429101d9937e3fc3/contracts/StateReceiver.sol#L41

---

id: core_concepts title: Core Concepts description: Bor is state chain in Polygon architecture keywords: - docs - matic - Core Concepts - polygon - state chain - architecture image: https://matic.network/banners/matic-network-16x9.png

import useBaseUrl from '@docusaurus/useBaseUrl';

Core Concepts

Bor is state chain in Polygon architecture. It is a fork of Geth https://github.com/ethereum/go-ethereum with new consensus called Bor.

Source: https://github.com/maticnetwork/bor

Consensus

Bor uses new improved consensus, inspired by Clique consensus

More details on consensus and specifications: Bor Consensus

Genesis

The genesis block contains all the essential information to configure the network. It's basically the config file for Bor chain. To boot up Bor chain, the user needs to pass in the location of the file as a param.

Bor uses `genesis.json` as Genesis block and params. Here is an example for Bor genesis `config` :

```
"config": {
    "chainId": 15001,
    "homesteadBlock": 1,
    "eip150Block": 0,
    "eip150Hash": "0x0000000000000000000000000000000000000000000000000000000000000000",
    "eip155Block": 0,
    "eip158Block": 0,
    "byzantiumBlock": 0,
    "constantinopleBlock": 0,
    "bor": {
      "period": 1,
      "producerDelay": 4,
      "sprint": 64,
      "validatorContract": "0x0000000000000000000000000000000000001000",
      "stateReceiverContract": "0x0000000000000000000000000000000000001001"
    }
  }
```

Config

Consensus specific config

EVM/Solidity as VM

Bor uses un-modified EVM as a VM for a transaction. Developers can deploy any contract they wish using the same Ethereum tools and compiler like `solc` without any changes.

MATIC as Native token (Gas token)

Bor has a MATIC token as a native token similar to ETH in Ethereum. It is often called the gas token. This token works correctly as to how ETH works currently on the Ethereum chain.

In addition to that, Bor provides an in-built wrapped ERC20 token for the native token (similar to WETH token), which means applications can use wrapped MATIC ERC20 token in their applications without creating their own wrapped ERC20 version of the Matic native token.

Wrapped ERC20 token is deployed at `0000000000000000000000000000000000001010` as `[MRC20.sol](https://github.com/maticnetwork/contracts/blob/develop/contracts/child/MRC20.sol)` on Bor as one of the genesis contracts.

Fees

Native token is used as fees while sending transaction on Bor. This prevents spam on Bor and provides incentives to Block Producers to run the chain for longer period and discourages bad behaviour.

A transaction sender defines `GasLimit` and `GasPrice` for each transaction and broadcasts it on Bor. Each producer can define how much minimum gas price they can accept using `--gas-price` while starting Bor node. If user-defined `GasPrice` on the transaction is the same or greater than producer defined gas price, the producer will accept the transaction and includes it in the next available block. This enables each producer to allow its own minimum gas price requirement.

Transaction fees will be deducted from sender's account in terms of Native token.

Here is the formula for transaction fees:

```
Tx.Fee = Tx.GasUsed * Tx.GasPrice
```

Collected fees for all transactions in a block are transferred to the producer's account using coinbase transfer. Since having more staking power increases your probability to become a producer, it will allow a validator with high staking power to collect more rewards (in terms of fees) accordingly.

Transfer receipt logs

The Matic token is no exception to that.

`LogTransfer` is a special log that is added to all compatible ERC20/721 tokens. Consider it as one 2-inputs-2-outputs UTXO for transfer. Here, `output1 = input1 - amount` and `output2 = input2 + amount` This allows fraud-proof contracts to verify a transfer of Matic ERC20 tokens (here, Native token) on the Ethereum chain.

```
/**
 * @param token    ERC20 token address
 * @param from     Sender address
 * @param to       Recipient address
 * @param amount   Transferred amount
 * @param input1   Sender's amount before the transfer is executed
 * @param input2   Recipient's amount before the transfer is executed
 * @param output1  Sender's amount after the transfer is executed
 * @param output2  Recipient's amount after the transfer is executed
 */
event LogTransfer(
    address indexed token,
    address indexed from,
```

```
    address indexed to,
    uint256 amount,
    uint256 input1,
    uint256 input2,
    uint256 output1,
    uint256 output2
);
```

Since, MATIC token is the native token and doesn't have Native ERC20 token, Bor adds receipt log for each transfer made for Native token using following Golang code. Source: https://github.com/maticnetwork/bor/blob/develop/core/state_transition.go#L241-L252

```
// addTransferLog adds transfer log into state
func addTransferLog(
    state vm.StateDB,
    eventSig common.Hash,

    sender,
    recipient common.Address,

    amount,
    input1,
    input2,
    output1,
    output2 *big.Int,
) {
    // ignore if amount is 0
    if amount.Cmp(bigZero) <= 0 {
        return
    }

    dataInputs := []*big.Int{
        amount,
        input1,
        input2,
        output1,
        output2,
    }

    var data []byte
    for _, v := range dataInputs {
        data = append(data, common.LeftPadBytes(v.Bytes(), 32)...)
    }

    // add transfer log
    state.AddLog(&types.Log{
        Address: feeAddress,
        Topics: []common.Hash{
            eventSig,
            feeAddress.Hash(),
            sender.Hash(),
            recipient.Hash(),
        },
        Data: data,
    })
}
```

Deposit native token

A user can receive native token by depositing MATIC tokens on Ethereum main-chain to `DepositManager` contract (deployed on Ethereum chain). Source: https://github.com/maticnetwork/contracts/blob/develop/contracts/root/depositManager/DepositManager.sol#L68

```
/**
 * Moves ERC20 tokens from Ethereum chain to Bor.
 * Allowance for the `_amount` tokens to DepositManager is needed before calling this function.
 * @param _token   Ethereum ERC20 token address which needs to be deposited
 * @param _amount  Transferred amount
 */
function depositERC20(address _token, uint256 _amount) external;
```

Using `depositERC20` tokens, users can move Matic ERC20 token (Native token) or any other ERC20 tokens from the Ethereum chain to Bor chain.

Withdraw native token

Withdraw from Bor chain to Ethereum chain works exactly like any other ERC20 tokens. A user can call `withdraw` function on ERC20 contract, deployed on Bor, at `0000000000000000000000000000000000001010` to initiate withdraw process for the same. Source: https://github.com/maticnetwork/contracts/blob/develop/contracts/child/MaticChildERC20.sol#L47-L61

```
/**
 * Withdraw tokens from Bor chain to Ethereum chain
 * @param amount     Withdraw amount
 */
function withdraw(uint256 amount) public payable;
```

In-built contracts (Genesis contracts)

Bor starts with three in-built contracts, often called genesis contracts. These contracts are available at block 0. Source: https://github.com/maticnetwork/genesis-contracts

These contracts are compiled using `solc --bin-runtime`. Example, following command emits compiled code for `contract.sol`

```
solc --bin-runtime contract.sol
```

Genesis contract is defined in `genesis.json`. When bor starts at block 0, it loads all contracts with the mentioned code and balance.

```
"0x0000000000000000000000000000000000001010": {
    "balance": "0x0",
    "code" : "0x..."
}
```

Below are the details for each genesis contract.

Bor validator set

Source: https://github.com/maticnetwork/genesis-contracts/blob/master/contracts/BorValidatorSet.sol

Deployed at: `0x0000000000000000000000000000000000001000`

`BorValidatorSet.sol` contract manages validator set for spans. Having a current validator set and span information into a contract allows other contracts to use that information. Since Bor uses producers from Heimdall (external source), it uses system call to change the contract state.

For first sprint all producers are defined in `BorValidatorSet.sol` directly.

`setInitialValidators` is called when the second span is being set. Since Bor doesn't support constructor for genesis contract, the first validator set needs to be set to `spans` map.

First span details are following:

```
firstSpan = {
  number: 0,
    startBlock: 0,
    endBlock: 255
}
```

Solidity contract definition:

```
contract BorValidatorSet {
  // Current sprint value
  uint256 public sprint = 64;

  // Validator details
  struct Validator {
    uint256 id;
    uint256 power;
    address signer;
  }

  // Span details
  struct Span {
    uint256 number;
    uint256 startBlock;
    uint256 endBlock;
  }

  // set of all validators
  mapping(uint256 => Validator[]) public validators;

  // set of all producers
  mapping(uint256 => Validator[]) public producers;

  mapping (uint256 => Span) public spans; // span number => span
  uint256[] public spanNumbers; // recent span numbers

    /// Initializes initial validators to spans mapping since there is no way to initialize through constructor for genesis
```

```
contract
    function setInitialValidators() internal

    /// Get current validator set (last enacted or initial if no changes ever made) with a current stake.
    function getInitialValidators() public view returns (address[] memory, uint256[] memory;

  /// Returns bor validator set at given block number
  function getBorValidators(uint256 number) public view returns (address[] memory, uint256[] memory);

  /// Proposes new span in case of force-ful span change
  function proposeSpan() external;

  /// Commits span (called through system call)
  function commitSpan(
    uint256 newSpan,
    uint256 startBlock,
    uint256 endBlock,
    bytes calldata validatorBytes,
    bytes calldata producerBytes
  ) external onlySystem;

  /// Returns current span number based on current block number
  function currentSpanNumber() public view returns (uint256);
}
```

`proposeSpan` can be called by any valid validator with zero fees. Bor allows `proposeSpan` transaction to be free transaction since it is part of the system.

`commitSpan` is being called through the system call.

State receiver

Source: https://github.com/maticnetwork/genesis-contracts/blob/master/contracts/StateReceiver.sol

Deployed at: `0x0000000000000000000000000000000000001001`

The `StateReceiver` contract provides a mechanism for receiving and storing state data from other contracts and notifying interested parties (i.e., contracts) of state changes. The state-sync mechanism allows for the transfer of state data from the Ethereum chain to Bor.

```
contract StateReceiver is System {
  using RLPReader for bytes;
  using RLPReader for RLPReader.RLPItem;

  uint256 public lastStateId;

  function commitState(uint256 syncTime, bytes calldata recordBytes) onlySystem external returns(bool success) {
    // parse state data
    RLPReader.RLPItem[] memory dataList = recordBytes.toRlpItem().toList();
    uint256 stateId = dataList[0].toUint();
    require(
      lastStateId + 1 == stateId,
      "StateIds are not sequential"
    );
    lastStateId++;

    address receiver = dataList[1].toAddress();
    bytes memory stateData = dataList[2].toBytes();
    // notify state receiver contract, in a non-revert manner
    if (isContract(receiver)) {
      uint256 txGas = 5000000;
      bytes memory data = abi.encodeWithSignature("onStateReceive(uint256,bytes)", stateId, stateData);
      // solium-disable-next-line security/no-inline-assembly
      assembly {
        success := call(txGas, receiver, 0, add(data, 0x20), mload(data), 0, 0)
      }
    }
  }

  // check if address is contract
  function isContract(address _addr) private view returns (bool){
    uint32 size;
    assembly {
      size := extcodesize(_addr)
    }
    return (size > 0);
  }
}
```

- `commitState` : Called by authorized contracts, this function updates the contract's state by parsing state data and checking its sequential order. If the data is from a contract, it calls the `onStateReceive` function on that contract.
- `isContract` : This function checks whether a given address belongs to a contract or not by checking its bytecode size, used in `commitState` .

MATIC ERC20 token

Source: https://github.com/maticnetwork/contracts/blob/develop/contracts/child/MaticChildERC20.sol

Deployed at: `0x0000000000000000000000000000000000001010`

This is special contract that wraps native coin (like $ETH in Ethereum) and provides an ERC20 token interface. Example: `transfer` on this contract transfers native tokens. `withdraw` method in ERC20 token allows users to move their tokens from Bor to Ethereum chain.

Note: This contract doesn't support `allowance` . This is same for every compatible ERC20 token contract.

```
contract MaticChildERC20 is BaseERC20 {
  event Transfer(address indexed from, address indexed to, uint256 value);

  uint256 public currentSupply;
  uint8 private constant DECIMALS = 18;

  constructor() public {}

  // Initializes state since genesis contract doesn't support constructor
  function initialize(address _childChain, address _token) public;

  /**
   * Deposit tokens to the user account
   * This deposit is only made through state receiver address
   * @param user   Deposit address
   * @param amount Withdraw amount
   */
  function deposit(address user, uint256 amount) public onlyOwner;

  /**
   * Withdraw amount to Ethereum chain
   * @param amount Withdraw amount
   */
  function withdraw(uint256 amount) public payable;

  function name() public pure returns (string memory) {
      return "Matic Token";
  }

  function symbol() public pure returns (string memory) {
      return "MATIC";
  }

  function decimals() public pure returns (uint8) {
      return DECIMALS;
  }

  /**
   * Total supply for the token.
   * This is 10b tokens, same as total Matic supply on Ethereum chain
   */
  function totalSupply() public view returns (uint256) {
      return 10000000000 * 10**uint256(DECIMALS);
  }

  /**
   * Balance of particular account
   * @param account Target address
   */
  function balanceOf(address account) public view returns (uint256) {
      return account.balance;
  }

  /**
   *  Function that is called when a user or another contract wants to transfer funds
   *  @param to Address of token receiver
   *  @param value Number of tokens to transfer
   *  @return Returns success of function call
   */
  function transfer(address to, uint256 value) public payable returns (bool) {
    if (msg.value != value) {
          return false;
    }
    return _transferFrom(msg.sender, to, value);
```

```
  }

  /**
   * This enables to transfer native token between users
   * while keeping the interface the same as that of an ERC20 Token
   * @param _transfer is invoked by _transferFrom method that is inherited from BaseERC20
   */
  function _transfer(address sender, address recipient, uint256 amount) internal {
    address(uint160(recipient)).transfer(amount);
    emit Transfer(sender, recipient, amount);
  }
}
```

System Call

Only system address, `2^160-2`, allows making a system call. Bor calls it internally with the system address as `msg.sender`. It changes the contract state and updates the state root for a particular block. Inspired from https://github.com/ethereum/EIPs/blob/master/EIPS/eip-210.md and https://wiki.parity.io/Validator-Set#contracts

System call is helpful to change state to contract without making any transaction.

Limitation: Currently events emitted by system call are not observable and not-included in any transaction or block.

Span Management

Span is a logically defined set of blocks for which a set of validators is chosen from among all the available validators. Heimdall will select the committee of producers out of all validators. The producers will include a subset of validators depending upon the number of validators in the system.

<img src={useBaseUrl("img/Bor/span-management.svg")} />

Propose Span Transaction

Type: **Heimdall transaction**

Source: https://github.com/maticnetwork/heimdall/blob/develop/bor/handler.go#L27

`spanProposeTx` sets validators' committee for a given `span` in case of successful transaction inclusion. One transaction for each span must be included in Heimdall. It is called `spanProposeTx` on Heimdall. `spanProposeTx` must revert if being sent frequently or there is no less than 33% stake change occurred within the current committee (for, given `span`).

`bor` module on Heimdall handles span management. Here is how Bor chooses producers out of all validators:

1. Bor creates multiple slots based on validators' power. Example: A with power 10 will have 10 slots, B with power 20 with have 20 slots.
2. With all slots, `shuffle` function shuffles them using `seed` and selects first `producerCount` producers. `bor` module on Heimdall uses ETH 2.0 shuffle algorithm to choose producers out of all validators. Each span `n` uses block hash of Ethereum (ETH 1.0) block `n` as `seed`. Note that slots based selection allows validators to get selected based on their power. The higher power validator will have a higher probability to get selected. Source: https://github.com/maticnetwork/heimdall/blob/develop/bor/selection.go

```
// SelectNextProducers selects producers for the next span by converting power to slots
// spanEligibleVals - all validators eligible for next span
func SelectNextProducers(blkHash common.Hash, spanEligibleVals []hmTypes.Validator, producerCount uint64) (selectedIDs []uint64,
err error) {
    if len(spanEligibleVals) <= int(producerCount) {
        for _, val := range spanEligibleVals {
            selectedIDs = append(selectedIDs, uint64(val.ID))
        }
        return
    }

    // extract seed from hash
    seed := helper.ToBytes32(blkHash.Bytes()[:32])
    validatorIndices := convertToSlots(spanEligibleVals)
    selectedIDs, err = ShuffleList(validatorIndices, seed)
    if err != nil {
        return
    }
    return selectedIDs[:producerCount], nil
}

// converts validator power to slots
func convertToSlots(vals []hmTypes.Validator) (validatorIndices []uint64) {
    for _, val := range vals {
        for val.VotingPower >= types.SlotCost {
            validatorIndices = append(validatorIndices, uint64(val.ID))
            val.VotingPower = val.VotingPower - types.SlotCost
        }
    }
```

```
    return validatorIndices
}
```

Commit span Tx

Type: **Bor transaction**

There are two way to commit span in Bor.

1. **Automatic span change**

   At the end of the current span, at last block of the last sprint, Bor queries the next span from Heimdall and set validators and producers for the next span using a system call.

   ```
   function commitSpan(
       bytes newSpan,
       address proposer,
       uint256 startBlock,
       uint256 endBlock,
       bytes validatorBytes,
       bytes producerBytes
    ) public onlySystem;
   ```

   Bor uses new producers as block producers for their next blocks.

2. **Force commit**

   Once the `span` proposed on Heimdall, the validator can force push span if span needs to be changed before the current span ends. A transaction to propose a `span` must be committed to Bor by any validator. Bor then updates and commits the proposed span at end of the current sprint using a system call.

State Management (State-sync)

State management sends the state from the Ethereum chain to Bor chain. It is called `state-sync`. This is a way to move data from the Ethereum chain to Bor chain.

<img src={useBaseUrl("img/Bor/state-managment.svg")} />

State sender

Source: https://github.com/maticnetwork/contracts/blob/develop/contracts/root/stateSyncer/StateSender.sol

To sync state sync, call following method **state sender contract** on Ethereum chain. The `state-sync` mechanism is basically a way to move state data from the Ethereum chain to Bor.

A user, who wants to move `data` from contract on Ethereum chain to Bor chain, calls `syncSate` method on `StateSender.sol`

```
contract StateSender {
    /**
     * Emits `stateSynced` events to start sync process on Ethereum chain
     * @param receiver    Target contract on Bor chain
     * @param data        Data to send
     */
    function syncState (
        address receiver,
        bytes calldata data
    ) external;
}
```

`receiver` contract must be present on the child chain, which receives state `data` once the process is complete. `syncState` emits `StateSynced` event on Ethereum, which is the following:

```
/**
 * Emits `stateSynced` events to start sync process on Ethereum chain
 * @param id                State id
 * @param contractAddress   Target contract address on Bor
 * @param data              Data to send to Bor chain for Target contract address
 */
event StateSynced (
    uint256 indexed id,
    address indexed contractAddress,
    bytes data
);
```

Once `StateSynced` event emitted on `stateSender` contract on the Ethereum chain, any validator sends `MsgEventRecord` transaction on Heimdall.

After confirmation of a tx on Heimdall, a validator proposes `proposeState` on Bor with the simple transaction and at end of the sprint, Bor commits and finalizes `state-sync` by calling `commitState` using a `system` call.

During `commitState` , Bor executes `onStateReceive` , with `stateId` and `data` as args, on target contract.

State Receiver Interface

`receiver` contract on Bor chain must implement following interface.

```
// IStateReceiver represents interface to receive state
interface IStateReceiver {
  function onStateReceive(uint256 stateId, bytes calldata data) external;
}
```

Only `0x0000000000000000000000000000000000001001` — `StateReceiver.sol` , must be allowed to call `onStateReceive` function on target contract.

Transaction Speed

Bor currently works as expected with ~2 to 4 seconds' block time with 100 validators and 4 block producers. After multiple stress testing with huge number of transactions, exact block time will be decided.

Using sprint-based architecture helps Bor to create faster bulk blocks without changing the producer during the current sprint. Having delay between two sprints gives other producers to receive a broadcasted block, often called as `producerDelay`

Note that time between two sprints is higher than normal blocks to buffer to reduce the latency issues between multiple producers.

Attacks

Censorship

Bor uses a very small set of producers to create faster blocks. It means it is prone to more censorship attacks than Heimdall. In order to deal with that, multiple testing will be done to find out the max number of producers for acceptable block time in the system.

Apart from that there are few attacks possible:

1. One producer is censoring the transaction

   In that case, the transaction sender can wait for the next producer's sprint and try to send the transaction again.

2. All validators are colluding with each-other and censoring particular transaction

   In this case, Polygon system will provide a way to submit a transaction on Ethereum chain and ask validators to include the transaction in next `x` checkpoints. If validators fail to include it during that time window, the user can slash the validators. Note that this is not currently implemented.

Fraud

Producers can include invalid transaction during their turn. It can be possible at multiple levels:

1. One producer is fraudulent

   If a producer includes invalid transaction at any height, other producers can create a fork and exclude that transaction since their valid node ignores invalid blocks

2. Span producers are fraudulent

   If other producers don't create a fork, other validators who are validating the block can forcefully change the span by creating their own fork. This is not currently implemented since it requires how Geth works internally. However, this is in our future roadmap.

3. All validators are fraudulent

   Assumption is that ⅔+1 validators must be honest to work this system correctly.

---

id: overview title: Overview description: The Bor node is basically the blockchain operator keywords: - docs - matic - polygon - bor - geth image: https://matic.network/banners/matic-network-16x9.png

Bor

The Bor node or the Block Producer implementation is basically the EVM-compatible blockchain operator. Currently, it is a basic Geth implementation with custom changes done to the consensus algorithm. However, this will be built from the ground up to make it lightweight and focused.

Block producers are chosen from the Validator set and are shuffled using historical Ethereum block hashes for the same purpose. However, we are exploring sources of randomness for this selection.--- id: getting-started title: Ethereum↔Polygon PoS Bridge sidebar_label: Overview description: A two-way transaction channel between Polygon and Ethereum. keywords: - docs - polygon - polygon wiki - crosschain bridge - polygon - ethereum - fx portal image: https://wiki.polygon.technology/img/polygon-logo.png

import useBaseUrl from '@docusaurus/useBaseUrl';

Polygon brings you a trustless two-way transaction channel between Polygon PoS and Ethereum by introducing the cross-chain bridge. With this users can transfer tokens across Polygon PoS without incurring third-party risks and market liquidity limitations. **The PoS Bridge is available on both Mumbai Testnet as**

well as Polygon PoS Mainnet. You can bridge assets to Polygon PoS using the Polygon Wallet Suite.

Polygon also offers the **zkEVM Bridge** which can also be used via the Polygon Wallet Suite. Read more about the zkEVM Bridge here. This document is more about the Polygon PoS Bridge.

Polygon PoS bridge provides a bridging mechanism that is near-instant, low-cost, and quite flexible.

There is no change to the circulating supply of your token when it crosses the bridge;

- Tokens that leave the Ethereum network are locked and the same number of tokens are minted on Polygon PoS as a pegged token (1:1).
- To move the tokens back to the Ethereum network, tokens are burned on Polygon PoS network and unlocked on Ethereum network during the process.

Additional Resources

- Introduction to Blockchain Bridges
- What are Cross-Chain Bridges

---

id: mintable-assets title: Mintable PoS Assets sidebar_label: Mintable assets description: Mint and create assets on the Polygon network with Fx-Portal. keywords: - docs - polygon wiki - polygon - mintable assets - mint image: https://wiki.polygon.technology/img/polygon-logo.png

import useBaseUrl from '@docusaurus/useBaseUrl';

Assets can be transferred to and from across the Ethereum and Polygon chain using the PoS bridge. These assets include ERC20, ERC721, ERC1155 and many other token standards. Most of the assets are pre-existing on Ethereum chain. But new assets can be created on the Polygon chain as well and moved back to Ethereum chain as and when required. This can save lots of gas and time that is spent on token minting on Ethereum.

Creation of assets on the Polygon chain is much easier and a more recommended approach. **These assets can be moved to Ethereum chain when required**. Such type of assets are called **Polygon mintable assets**.

In the case of Polygon Mintable tokens, assets are created on the Polygon network. When a Polygon minted asset has to be moved to Ethereum, the asset has to be burned first and then a proof of this burn transaction has to be submitted on the Ethereum chain. The recommended way to make use of Polygon Mintable token capabilities is by utilizing the fx-portal.

id: submit-mapping-request title: Mapping Tokens sidebar_label: Mapping tokens description: A guide on how to map tokens between Ethereum and Polygon Chains using the PoS Bridge keywords: - docs - polygon wiki - token mapping - pos bridge - polygon - goerli - ethereum - testnet - mainnet image: https://wiki.polygon.technology/img/polygon-logo.png

import useBaseUrl from '@docusaurus/useBaseUrl';

Mapping is necessary in order to transfer your assets to and from Ethereum and Polygon PoS. We offer two bridges to do the same. More details on the bridge can be understood here.

:::info

The Polygon PoS bridge is available for both Polygon Mainnet as well as Mumbai Testnet.

:::

Steps to submit a mapping request

In order to map tokens between Ethereum and Polygon PoS, you may use the Polygon Token Mapper. Open the link and click on the **Map New Token** button on the top right corner to initiate a new mapping request.

<img src={useBaseUrl("img/token-mapping/mapping-tool.png")} />

Step 1 → Choose the network on which you want to map your token. You may choose **Goerli-Mumbai** for Testnet, and **Ethereum-Polygon PoS** for the Mainnet.

Step 2 → Select the type of token you are mapping - **ERC20**, **ERC721**, or **ERC1155**.

Step 3 → Enter your **Ethereum/Goerli** token address in the **Ethereum Token Address** field. Make sure your token contract code has been verified on the **Ethereum/Goerli** blockchain explorers.

Step 4 → After adding the **Ethereum Token Address**, the corresponding fields viz. **Token Name, Token Symbol, and Token Decimal** will be automatically populated with the contract details.

Step 5 → Now, click on the **Begin Mapping** button to initiate the mapping process. As this involves an Ethereum transaction, you will need to connect your wallet to proceed.

Step 6 → You will be shown a review modal with the token information and the estimated gas fees to complete the mapping. Verify the details and initiate the mapping transaction by selecting the **Pay Gas Fee To Map** button.

After confirming the transaction from your wallet, you have to wait for the transaction to get completed on Ethereum. Once the transaction is completed, you will be shown the success modal with your child token address on the Polygon PoS network. You can continue to verify the mapping by checking the generated child token address on Polygonscan.

For a successful Mainnet mapping, you may provide your token details here to be added on the **Polygon Token List**.

:::tip Want a Custom Token Mapping?

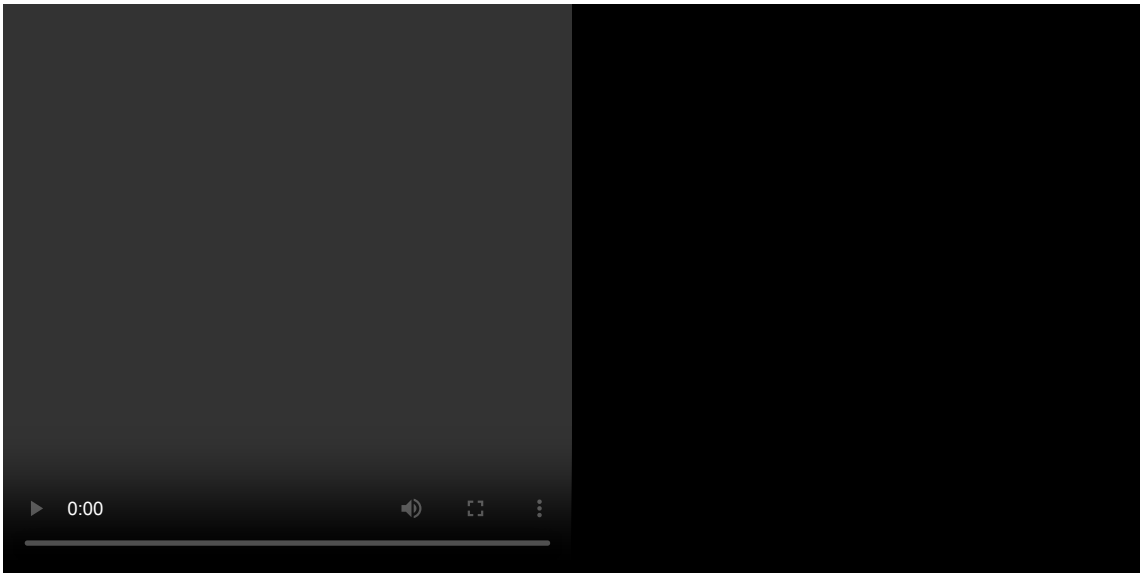
In case you want to customize your token mapping, you can visit our **FxPortal** documentation and use the information provided to build your FxTunnel implementation and map tokens.

While using the **default FxPortal** tunnels, you can't modify the child contract code. In case you need a **custom child token**, you must go for your own **custom FxTunnel**.

:::

Video Guide

Here is a quick video tutorial on how to map tokens between **Ethereum Goerli ↔ Polygon Mumbai Testnet**:



--- id: widget title: Wallet Widget sidebar_label: Wallet Widget description: "UI tools to execute bridge transactions." keywords: - docs - matic image: https://matic.network/banners/matic-network-16x9.png --- import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem';

Wallet widget is a UI tool which can be embedded in any web application for executing bridge transactions - Deposit & Withdraw.

Every widget is identified by an unique name which you can get from Widget dashboard .

Widget dashboard

Widget can be created from the widget dashboard page in the wallet application. It allows the user to create a new widget with some customisable options.

Once the widget is created, You can copy code snippet and add it in your application or use the widget name and configure by yourself.

Here is link to widget dashboard -

- mainnet - https://wallet.polygon.technology/widget-dashboard
- testnet - https://wallet-dev.polygon.technology/widget-dashboard

Install

Widget is exported as javascript library and available as npm package.

```
npm i @maticnetwork/wallet-widget
```

Examples

We have created examples for different framework and tools to help you with the development. All examples are present at - https://github.com/maticnetwork/wallet-widget-example

How to use

With target

Consider you have a button in your app and you want to show widget when clicked on that button -

```
<button id="btnMaticWidget"></btn>
```

```
import { Widget } from "@maticnetwork/wallet-widget";

var widget = new Widget({
    appName: "<widget name>", //widget name from dashboard
```

```
    target: '#btnMaticWidget', // element selector for showing widget on click
    network: 'mainnet' // network to be used - testnet or mainnet
});
```

Create widget whenever you are ready. It is best to call create function after document is loaded.

```
await widget.create();
```

widget is created, now click on your button and widget will be shown.

Without target

```
import { Widget } from "@maticnetwork/wallet-widget";

var widget = new Widget({
    appName: "<widget name>", //widget name from dashboard
    network: 'mainnet' // network to be used - testnet or mainnet
});

await widget.create();
```

widget is now created, but in order to show the widget - you will have to call `show` API.

```
widget.show();
```

Similarly you can hide the widget, by calling `hide` API.

```
widget.hide();
```

Important Note 👉

1. Based on network "testnet" or "mainnet", you need to create your app on respective dashboard. We recommend to create app with same name on both testnet & mainnet, so that you don't have any issue when you are changing network.
2. Wallet widget is UI Library and on different website it will look different & might have some issues like - colors, responsiveness etc. So please spend some time on testing and customizing. In case of any help needed - please reach out to support team.
3. Wallet widget is full screen in mobile devices but you can customize it by `style` configuration.

Configuration

Configuration can be supplied in Widget constructor.

Available configuration are

- **target** : string - CSS selector for showing widget on click of element. For example, "#btnMaticWidget" will be the target in the code below.

```
<button id="btnMaticWidget">Matic widget</button>
```

- **network** : string - network to be used. Two options are available - 'testnet' or 'mainnet'.
- **width** : number - Width of the widget
- **height** : number - Height of the widget
- **autoShowTime** : number - Auto show widget after specified time in millisecond
- **appName** : string - name of your app, this can be retrieved on widget dashboard.
- **position** : string - Sets the position of the widget. The available options are -
  - center
  - bottom-right
  - bottom-left
- **amount** : string - Prefill the amount in text box
- **page** : string - select the page. Available options are - `withdraw`, `deposit`.
- **overlay** : boolean - show overlay when widget is opened. By default it is false.
- **style** : object - apply some css styles to the widget.

```
var widget = new MaticWidget({
    appName: "<your app id>", //appName from dashboard
    target: '#btnMaticWidget', // element selector for showing widget on click
    network: 'testnet' // network to be used - testnet or mainnet,
    style:{
      color:'red'
    }
});
```

Events

Widget emits some events which can be used to know what is happening inside the application.

Subscribe to events

```
widget.on('load',()=>{
  console.log('widget is loaded');
})
```

Unsubscribe to events

```
widget.off('load',<callback>)
```

> *Callback should be same as what was used to subscribe the event. So its better to store the callback in a variable.* `

List of events:

- **load** - Widget is loaded
- **close** - Widget is closed
- **approveInit** - Approval transaction is initialized
- **approveComplete** - Approval transaction is completed
- **approveError** - Approval transaction failed due to some error, or the user denied the transaction on Metamask
- **depositInit** - Deposit transaction is initialized
- **depositComplete** - Deposit transaction is completed
- **depositError** - Deposit transaction failed due to some error, or the user denied the deposit complete transaction on Metamask
- **burnInit** - Withdrawal burn transaction is initialized
- **burnComplete** - Withdrawal burn transaction is completed
- **confirmWithdrawInit** - Withdrawal is checkpointed and confirm transaction is initialized
- **confirmWithdrawComplete** - Withdrawal confirm transaction in completed
- **confirmWithdrawError** - Withdrawal confirm transaction failed due to some error, or the user denied the withdrawal confirm transaction on Metamask
- **exitInit** - Withdrawal exit transaction is initialized
- **exitComplete** - Withdrawal exit transaction is completed
- **exitError** - Withdrawal exit transaction failed due to some error, or the user denied the withdrawal exit transaction on Metamask

APIS

- **show** - show the widget

```
widget.show()
```

- **hide** - hide the widget

```
widget.hide()
```

- **on** - subscribe to events

```
widget.on('<event name>', callback)
```

- **off** - unsubscribe to events

```
widget.off('<event name>', callback)
```

---

id: ethereum-to-matic title: Transfer data from Ethereum to Polygon description: Transfer state or data from Ethereum to Polygon via Contracts keywords: - docs - matic image: https://matic.network/banners/matic-network-16x9.png

The mechanism to natively read Ethereum data from Polygon EVM chain is that of 'State Sync'. In other words, this mechanism enables transfer of arbitrary data from Ethereum chain to Polygon chain. The procedure that makes it possible is: Validators on the Heimdall layer are listening for a particular event — `StateSynced` from a Sender contract, as soon as the event is picked, the `data` that was passed in the event is written on the Receiver contract. Read more here.

The Sender and Receiver contract are required to be mapped on Ethereum — StateSender.sol needs to be aware of each sender and receiver. If you'd like to get the mapping done, please request a mapping here.

In the following walkthrough, we'll be deploying a Sender contract on Goerli (Ethereum testnet) and a Receiver contract on Mumbai (Polygon's testnet) and then we'll be sending data from Sender and reading data on Receiver via web3 calls in a node script.

1. Deploy Sender contract

The sole purpose of Sender contract is to be able to call syncState function on the StateSender contract — which is Matic's state syncer contract - the StateSynced event of which Heimdall is listening to.

Deployed at:

`0xEAa852323826C71cd7920C3b4c007184234c3945` on Goerli

`0x28e4F3a7f651294B9564800b2D01f35189A5bFbE` on Ethereum Mainnet

To be able to call this function, let's first include it's interface in our contract:

```
// Sender.sol

pragma solidity ^0.5.11;

contract IStateSender {
  function syncState(address receiver, bytes calldata data) external;
  function register(address sender, address receiver) public;
}

...
```

Next, let's write our custom function that takes in the data we'd like to pass on to Polygon and calls syncState

```
function sendState(bytes calldata data) external {
    states = states + 1 ;
    IStateSender(stateSenderContract).syncState(receiver, data);
}
```

In the above function, `stateSenderContract` is the address of the StateSender on the network you'll be deploying `Sender` on. (eg., we'll be using `0xEAa852323826C71cd7920C3b4c007184234c3945` for Goerli), and `receiver` is the contract that will receive the data we send from here.

It is recommended to use constructors to pass in variables, but for the purpose of this demo, we'll simply hardcode these two addresses:

Following is how our Sender.sol looks like:

```
// sender.sol

pragma solidity ^0.5.11;

contract IStateSender {
  function syncState(address receiver, bytes calldata data) external;
  function register(address sender, address receiver) public;
}

contract sender {
  address public stateSenderContract = 0xEAa852323826C71cd7920C3b4c007184234c3945;
  address public receiver = 0x83bB46B64b311c89bEF813A534291e155459579e;

  uint public states = 0;

  function sendState(bytes calldata data) external {
    states = states + 1 ;
    IStateSender(stateSenderContract).syncState(receiver, data);
  }

}
```

We're using a simple `states` counter to keep track of the number of states sent via the Sender contract.

Use Remix to deploy the contract and keep a note of the address and ABI.

2. Deploy Receiver contract

Receiver contract is the one that is invoked by a Validator when the `StateSynced` event is emitted. The Validator invokes the function `onStateReceive` on the receiver contract to submit the data. To implement it, we first import StateReceiver interface and write down our custom logic — to interpret the transferred data inside onStateReceive.

Following is how our Receiver.sol looks like:

```
// receiver.sol

pragma solidity ^0.5.11;

// IStateReceiver represents interface to receive state
interface IStateReceiver {
  function onStateReceive(uint256 stateId, bytes calldata data) external;
}

contract receiver {

  uint public lastStateId;
  bytes public lastChildData;
```

```
  function onStateReceive(uint256 stateId, bytes calldata data) external {
    lastStateId = stateId;
    lastChildData = data;
    }

}
```

The function simply assigns the last received State Id and data to variables. StateId is a simple unique reference to the transferred state (a simple counter).

Deploy your Receiver.sol on Polygon's testnet and keep a note of the address and ABI

3. Getting your Sender and Receiver mapped

You can either use the already deployed addresses (mentioned above) for sender and receiver, or deploy your custom contracts and request a mapping done here: https://mapper.polygon.technology/

4. Sending and Receiving data

Now that we have our contracts in place and mapping done, we'll be writing a simple node script to send arbitrary hex bytes, receive them on Polygon and interpret the data!

4.1 Setup your script

We'll first initialise our web3 objects, wallet to make the transactions and contracts

```
// test.js

const Web3 = require('web3')
const Network = require("@maticnetwork/meta/network")

const network = new Network ('testnet', 'mumbai')

const main = new Web3(network.Main.RPC)
const matic = new Web3 (network.Matic.RPC)

let privateKey = `0x...` // add or import your private key

matic.eth.accounts.wallet.add(privateKey)
main.eth.accounts.wallet.add(privateKey)

let receiverAddress = `<RECEIVER_CONTRACT_ADDRESS>`
let receiverABI = `` // insert or import ABI
let senderAddress = `<SENDER_CONTRACT_ADDRESS>`
let senderABI = `` // insert of import the ABI

let sender = new main.eth.Contract(JSON.parse(senderABI), senderAddress)
let receiver = new matic.eth.Contract(JSON.parse(receiverABI), receiverAddress)
```

We're using @maticnetwork/meta package for the RPCs, the package isn't a requirement to run the script.

`matic` and `main` objects refer to the web3 object initialised with Polygon's and Ropsten's RPC respectively.

`sender` and `receiver` objects refer to the contract objects of Sender.sol and Receiver.sol that we deployed in Step 1 and 2.

4.2 Sending data

Next, let's setup our functions to create bytestring of the data and send it via Sender contract:

```
// data to sync
function getData(string) {
  let data = matic.utils.asciiToHex(string);
  return data
}

// send data via sender
async function sendData (data) {
  let r = await sender.methods
    .sendState (getData(data))
    .send({
      from: main.eth.accounts.wallet[0].address,
      gas: 8000000
    })
  console.log('sent data from root, ', r.transactionHash)
}
```

Calling `getData` will convert an ascii string (eg., `Hello World !`) to a string of bytes (eg., `0x48656c6c6f20576f726c642021`); while the function `sendData` takes in `data` (an ascii string), calls `getData` and passes on the bytestring to sender contract

4.3 Receiving data

Next, we'll be checking for received data on Receiver.sol.

It should take ~7-8 minutes for the state sync to execute.

Add the following functions to check (a) number of sent states from Sender and (b) Last received state on Receiver.

```
// check `states` variable on sender
async function checkSender () {
  let r = await sender.methods
    .states()
    .call()
  console.log('number of states sent from sender: ', r)
}

// check last received data on receiver
async function checkReceiver () {
  let r = await receiver.methods
    .lastStateId()
    .call()
  let s = await receiver.methods
    .lastChildData()
    .call()
  console.log('last state id: ', r, 'and last data: ', s)
  console.log('interpreted data: ', getString(s))
}
```

the function `checkReceiver` simply calls the variables we defined in the contract — which would be set as soon as the Validator calls `onStateReceive` on the contract. The `getString` function simply interprets the bytestring (converts it back to ascii)

```
function getString(data) {
  let string = matic.utils.hexToAscii(data);
  return string
}
```

Finally, we'll write up a method to execute our functions:

```
async function test() {
    await sendData ('Sending a state sync! :) ')
    await checkSender ()
    await checkReceiver ()
}
```

4.4 Putting it all together!

This is how our test script looks like:

```
// test.js

const Web3 = require('web3')
const Network = require("@maticnetwork/meta/network")

const network = new Network ('testnet', 'mumbai')

const main = new Web3(network.Main.RPC)
const matic = new Web3 (network.Matic.RPC)

let privateKey = `0x...`
matic.eth.accounts.wallet.add(privateKey)
main.eth.accounts.wallet.add(privateKey)

let receiverAddress = `<RECEIVER_CONTRACT_ADDRESS>`
let receiverABI = ``
let senderAddress = `<SENDER_CONTRACT_ADDRESS>`
let senderABI = ``

let sender = new main.eth.Contract(JSON.parse(senderABI), senderAddress)
let receiver = new matic.eth.Contract(JSON.parse(receiverABI), receiverAddress)

// data to sync
function getData(string) {
  let data = matic.utils.asciiToHex(string);
  return data
}

function getString(data) {
  let string = matic.utils.hexToAscii(data);
  return string
```

```
}

// console.log(getData('Sending a state sync! :) '))

async function sendData (data) {
  let r = await sender.methods
    .sendState (getData(data))
    .send({
      from: main.eth.accounts.wallet[0].address,
      gas: 8000000
    })
  console.log('sent data from root, ', r.transactionHash)
}

async function checkSender () {
  let r = await sender.methods
    .states()
    .call()
  console.log('number of states sent from sender: ', r)
}

async function checkReceiver () {
  let r = await receiver.methods
    .lastStateId()
    .call()
  let s = await receiver.methods
    .lastChildData()
    .call()
  console.log('last state id: ', r, 'and last data: ', s)
  console.log('interpreted data: ', getString(s))
}

async function test() {
    await sendData ('Hello World !')
    await checkSender ()
    // add a timeout here to allow time gap for the state to sync
    await checkReceiver ()
}

test()
```

4.5 Let's run the script

Successful execution of the above script provide an output as:

```
$ node test
> sent data from root 0x4f64ae4ab4d2b2d2dc82cdd9ddae73af026e5a9c46c086b13bd75e38009e5204
number of states sent from sender: 1
last state id: 453 and last data: 0x48656c6c6f20576f726c642021
interpreted data: Hello World !
```

---

id: fx-portal title: FxPortal Overview sidebar_label: Overview description: Transfer state or data from Ethereum to Polygon without mapping using FxPortal. keywords: - docs - polygon wiki - polygon - FxPortal - ethereum to polygon image: https://wiki.polygon.technology/img/polygon-logo.png

import useBaseUrl from '@docusaurus/useBaseUrl';

The usual mechanism to natively read Ethereum data from Polygon is using **State Sync**. This enables the transfer of arbitrary data from Ethereum to Polygon. However, this approach also requires mapping of the root and child contracts if the default interface cannot be used. FxPortal offers an alternative where ERC standardized tokens can be deployed without any mapping involved, simply using the deployed base FxPortal contracts.

What is FxPortal?

It is a powerful yet simple implementation of the Polygon state sync mechanism. The Polygon PoS bridge is built on the same architecture. The code in the examples folder are some examples of usage. You can easily use these examples to build your own implementations or own custom bridge which allows any state-sync without mapping.

You can check out the GitHub repository for contracts and examples.

How does it work?

FxChild and FxRoot are the main contracts on which FxPortal works. It calls and passes data to user-defined methods on the other chain without any mapping using the state sync mechanism. To use the deployed main contracts, you can implement FxPortal's base contracts in the smart contracts you deploy - FxBaseRootTunnel and FxBaseChildTunnel. By building on these contracts, your deployed contracts will be able to communicate with each other using the data tunnel mechanism.

Otherwise, you can choose to map your tokens with the already deployed tunnel contracts. Default FxTunnel deployment details for Polygon Mainnet and Mumbai Testnet are as follows:

- Polygon Mainnet
- Mumbai Testnet

Look for the keyword `FxPortalContracts` in the above links to find all the default tunnel contracts and other important FxPortal contract deployments.

Do I need a Custom FxTunnel Implementation ?

You must go for a **custom FxTunnel implementation** only if the default tunnel implementations do not align with your use case. When you use the default FxPortal tunnels, you can not modify the child contract code. The bytecode for the child token contract is always fixed and always remains the same for the default FxTunnel deployments. In case you need a custom child token, you must go for your own custom FxTunnel, and reading the next part will guide you more in deploying your own custom FxTunnels.

It is highly recommended to read and understand FxPortal State Transfer before you read the upcoming section. Each of these upcoming sections will have example tunnel contract links attached to it. These examples can be taken as a reference while building your own custom fx-tunnels.

ERC20 Transfer

The child and root tunnel contracts enable the deposit of tokens on the root chain and withdrawal on the child chain.

`FxERC20RootTunnel`

- `mapToken(address rootToken)` : You can call the function on the deployed contract to map your ERC20 token and create a corresponding child token on the child chain.
- `deposit(address rootToken, address user, uint256 amount, bytes memory data)` : call `deposit()` method with the address of the mapped token, the address that can withdraw with a corresponding amount (along with data if needed). You must have approved the contract using the standard ERC20 `approve` function to spend your tokens first.

`FxERC20ChildTunnel`

- `withdraw(address childToken, uint256 amount)` : The address assigned in `deposit()` can withdraw all the amount of child token using this function. They will receive the child token created when first mapped.
- `rootToChildToken` : This public variable contains the root token to child token mapping. You can query the mapping with the address of the root token to know the address of the deployed child token.

From Ethereum → Polygon

1. Deploy your own ERC20 token on the root chain. You will need this address later.
2. Approve the tokens for transfer by calling the `approve()` function of the root token with the address of the root tunnel and the amount as the arguments.
3. Proceed to call `deposit()` with the address of the receiver and amount on the root chain to receive the equivalent child token on the child chain. This will also map the token automatically. Alternatively, you can call `mapToken()` first before depositing.
4. After mapping, you should now be able to execute cross-chain transfers using the `deposit` and `withdraw` functions of the tunnel.

:::note

After you have performed `deposit()` on the root chain, it will take 22-30 minutes for state sync to happen. Once state sync happens, you will get the tokens deposited at the given address.

:::

From Polygon → Ethereum

1. Proceed to call `withdraw()` with the respective token address and amount as arguments on the child contract to move the child tokens back to the designated receiver on the root chain. **Note the tx hash** as this will be used to generate the burn proof.

2. You can find the steps to complete the withdrawal here.

ERC721 Transfer

In case you need an example, please check out this ERC721 Root and Child Tunnels guide.

From Ethereum → Polygon

1. Deploy your own ERC721 token on the root chain. You will need this address later.
2. Approve the tokens for transfer by calling the `approve()` function of the root token with the address of the root tunnel and the token ID as the arguments.
3. Proceed to call `deposit()` with the address of the receiver and token ID on the root chain to receive the equivalent child token on the child chain. This will also map the token automatically. Alternatively, you can call `mapToken()` first before depositing.

:::note

After you have performed `deposit()` on the root chain, it will take 22-30 minutes for state sync to happen. Once state sync happens, you will get the tokens deposited at the given address.

:::

From Polygon → Ethereum

1. Proceed to call `withdraw()` with the respective token address and token id as arguments on the child contract to move the child tokens back to the designated receiver on the root chain. **Note the tx hash** will be used to generate the burn proof.

2. You can find the steps to complete the withdrawal here.

ERC1155 Transfer

In case you need an example, please check out this ERC1155 Root and Child Tunnels guide.

`FxERC1155RootTunnel`

- `mapToken(rootToken)` : Used to map your root ERC1155 token to child chain
- `deposit(rootToken, user, id, amount, data)` : Function used to deposit root tokens to child chain
- `depositBatch(rootToken, user, ids, amounts, bytes memory data)` : Used for multiple token ids and corresponding amounts
- `receiveMessage(inputData)` : To be called after burn proof has been generated with the payload as `inputData`

`FxERC1155ChildTunnel`

- `withdraw(childToken, id, amount, data)` : Used to withdraw token from Polygon to Ethereum
- `withdrawBatch(childToken, ids, amounts, data)` : Same as withdraw but for withdrawing multiple token ids

From Ethereum → Polygon

1. Deploy your ERC1155 token on the root chain. You will need this address later.
2. Call `setApprovalForAll(operator, approved)` on the deployed token with `FxERC1155RootTunnel` address as `operator` to allow `FxERC1155RootTunnel` to transfer your tokens to `FxERC1155ChildTunnel` on Polygon.
3. Call `mapToken()` on `FxERC1155RootTunnel` with your deployed token's address as `rootToken` . This will send a message to `FxERC1155ChildTunnel` instructing it to deploy and map the ERC1155 token on Polygon. To query your child token address, call `rootToChildToken` on `FxERC1155ChildTunnel` .
4. Call `deposit()` on `FxERC1155RootTunnel` with the address of the token on Ethereum as `rootToken` , receiver as `user` , token id as `id` and the amount as `amount` . Alternatively, you can also call `depositBatch()` for multiple token ids.

:::note

After you have performed `deposit()` on the root chain, it will take 22-30 minutes for state sync to happen. Once state sync happens, you will get the tokens deposited at the given address.

:::

From Polygon → Ethereum

1. Call `withdraw()` on `FxERC1155ChildTunnel` with the address of the child token deployed on Polygon as the `childToken` and the token id as `id` (the child token address can be queried from `rootToChildToken` mapping). Alternatively, you can also call `withdrawBatch()` for multiple token ids and corresponding amounts. **Note the tx hash** will be used to generate the burn proof.

2. You can find the steps to complete the withdrawal here.

Withdraw Tokens on the Root Chain

:::info

After you have performed `withdraw()` on the child chain, it will take 30-90 minutes for a checkpoint to happen. Once the next checkpoint includes the burn transaction, you can withdraw the tokens on the root chain.

:::

1. Generate the burn proof using the **tx hash** and **MESSAGE_SENT_EVENT_SIG**. To generate the proof, you can either use the proof generation API hosted by Polygon or you can also spin up your own proof generation API by following the instructions here.

The proof generation endpoint hosted by Polygon is available here.

- `burnTxHash` is the transaction hash of the `withdraw()` transaction you initiated on Polygon.
- `eventSignature` is the event signature of the event emitted by the `withdraw()` function. The event signature for the MESSAGE_SENT_EVENT_SIG is `0x8c5261668696ce22758910d05bab8f186d6eb247ceac2af2e82c7dc17669b036` .

The proof generation API usage examples for the Mainnet and Testnet are as follows:-

→ Polygon Mainnet Proof generation

→ Mumbai Testnet Proof generation

2. Feed the generated payload as the argument to `receiveMessage()` in the respective root tunnel contract on Goerli or Ethereum.

Mintable ERC-20 Transfer

In case you need an example, please check out this Mintable ERC20 Root and Child Tunnels guide.

:::info

In the case of Mintable Token FxTunnels, the child token gets deployed first and the root token is deployed only when the first withdraw/exit process is completed. The root token contract address can be pre-determined right after the child contract is deployed, but the mapping will technically exist only when the first withdrawal/exit is completed.

:::

`FxMintableERC20RootTunnel`

- `deposit(address rootToken, address user, uint256 amount, bytes memory data)` : To deposit tokens from Ethereum to Polygon
- `receiveMessage(bytes memory inputData)` : Burn proof to be fed as the `inputData` to receive tokens on the root chain

`FxMintableERC20ChildTunnel`

- `deployChildToken(uint256 uniqueId, string memory name, string memory symbol, uint8 decimals)` : To deploy an ERC20 token on the Polygon network
- `mintToken(address childToken, uint256 amount)` : Mint a particular amount of tokens on Polygon
- `withdraw(address childToken, uint256 amount)` : To burn tokens on the child chain in order to withdraw on the root chain

Minting Tokens on Polygon

1. Call the `deployChildToken()` on `FxMintableERC20ChildTunnel` and pass the necessary token info as parameters. This emits a `TokenMapped` event which contains the `rootToken` and `childToken` addresses. Note these addresses.
2. Call `mintToken()` on `FxMintableERC20ChildTunnel` to mint tokens on the child chain.
3. Call `withdraw()` on `FxMintableERC20ChildTunnel` to withdraw tokens from Polygon. Note the transaction hash as this will come in handy to generate the burn proof.
4. You can find the steps to complete the withdrawal here.

Withdrawing Tokens on Ethereum

Feed the generated burn proof as the argument to `receiveMessage()` in `FxMintableERC20RootTunnel`. After this, the token balance would be reflected on the root chain.

Deposit Tokens back to Polygon

1. Make sure you approve `FxMintableERC20RootTunnel` to transfer your tokens.
2. Call `deposit()` in `FxMintableERC20RootTunnel` with the `rootToken` as address of root token and `user` as the recipient.
3. Wait for the state sync event (22-30 mins). After this, you can query the target recipient's balance on the child chain.

The **ERC721** and **ERC1155** Mintable FxTunnel examples are as follows :-

- FxMintableERC721Tunnels
- FxMintableERC1155Tunnels

Example Deployments

Goerli

- Checkpoint Manager: 0x2890bA17EfE978480615e330ecB65333b880928e
- Dummy ERC20 token: 0xe9c7873f81c815d64c71c2233462cb175e4765b3
- FxERC20RootTunnel: 0x3658ccFDE5e9629b0805EB06AaCFc42416850961
- FxMintableERC20RootTunnel: 0xA200766a7D64E54611E2D232AA6c1f870aCb63c1
- Dummy ERC721 token: 0x73594a053cb5ddDE5558268d28a774375C4E23dA
- FxERC721RootTunnel: 0xF9bc4a80464E48369303196645e876c8C7D972de
- Dummy ERC1155 Token: 0x1906d395752FE0c930f8d061DFEb785eBE6f0B4E
- FxERC1155RootTunnel : 0x48DE785970ca6eD289315036B6d187888cF9Df48

Mumbai

- FxERC20: 0xDDE69724AeFBdb084413719fE745aB66e3b055C7
- FxERC20ChildTunnel: 0x9C37aEbdb7Dd337E0215BC40152d6689DaF9c767
- FxMintableERC20ChildTunnel: 0xA2C7eBEf68B444056b4A39C2CEC23844275C56e9
- Child token dummy ERC20: 0x346d21bc2bD3dEE2d1168E1A632b10D1d7B9c0A
- FxERC721: 0xf2720927E048726267C0221ffA41A88528048726
- FxERC721ChildTunnel: 0x3658ccFDE5e9629b0805EB06AaCFc42416850961
- FxERC1155: 0x80be8Cf927047A40d3f5791BF7436D8c95b3Ae5C
- FxERC1155ChildTunnel: 0x3A0f90D3905601501652fe925e96d8B294243Efc

The corresponding Mainnet deployments can be found here. Look for the keyword `FxPortalContracts` to find all the default tunnel contracts and other important FxPortal contract deployments. You can make use of the `maticnetwork/meta` package to access the contract addresses and ABIs.

Contract Addresses

Mumbai Testnet

Contract	Deployed address
FxRoot (Goerli)	`0x3d1d3E34f7fB6D26245E6640E1c50710eFFf15bA`
FxChild (Mumbai)	`0xCf73231F28B7331BBe3124B907840A94851f9f11`

Polygon Mainnet

Contract	Deployed address
FxRoot (Ethereum Mainnet)	`0xfe5e5D361b2ad62c541bAb87C45a0B9B018389a2`
FxChild (Polygon Mainnnet)	`0x8397259c983751DAf40400790063935a11afa28a`

---

id: fx-example title: Deployment Example description: Transfer state or data from Ethereum to Polygon without mapping using FxPortal. keywords: - docs - polygon wiki - polygon - FxPortal - ethereum to polygon image: https://wiki.polygon.technology/img/polygon-logo.png

Overview

The contracts in the following example consist of two main parts, one for the root chain (Goerli in this case), and one for the child chain (Mumbai). The Goerli contract is responsible for mapping tokens from the root chain to the child chain and sending deposits to the child chain. The Mumbai contract is responsible for handling incoming messages from the Goerli contract, minting child tokens, and processing child-to-root withdrawals.

Here is an overview of the following contracts used:

- **`SafeToken`** (Goerli): This contract deploys a simple ERC20 token that can be used to test the FxPortal functionality.
- **`FxERC20`** (Goerli): This contract template will create new child tokens on the Mumbai network. The contract is initialized with the address of the FxPortal contract and the address of the corresponding token on the root network. It contains a mint function that the root network contract to mint new child tokens can use.
- **`FxERC20`** (Mumbai): This contract is a child token minted by the root network contract. It is initialized with the address of the FxPortal contract, the address of the corresponding token on the root network, and the token's name, symbol, and decimals. It contains a burn function that can be used to burn child tokens.
- **`FxERC20RootTunnel`** (Goerli): This contract is responsible for mapping tokens from the root network to the child network and sending deposits to the child network. When a new token is mapped, the contract deploys a new child token contract using the `FxERC20` template and then maps the root token to the child token. The contract sends a message to the child network containing the deposit details when a deposit is made.
- **`FxERC20ChildTunnel`** (Mumbai): This contract is responsible for processing messages from the root network and minting new child tokens, as well as handling child-to-root withdrawals. When a new token is mapped, the contract deploys a new child token contract using the `FxERC20` template and then maps the root token to the child token. When a deposit message is received, the contract mints new child tokens and calls the `onTokenTransfer` function on the receiver (if it is a contract). When a withdrawal is made, the contract burns the child tokens and sends a message to the root network containing the withdrawal details.

Prerequisites

- Goerli and Mumbai testnet accounts with some testnet ETH and MATIC tokens, respectively
- Metamask wallet

Steps

1. Deploy the SafeToken contract on the Goerli testnet (using Remix or Truffle). This will be the ERC20 token to test the FxPortal functionality.

2. Deploy an FxERC20 contract on the Goerli network. This contract will be used as a template to create child tokens on the Mumbai network. The FxERC20 contract should be initialized with the address of the FxPortal contract and the address of the corresponding token on the root network.

3. Deploy an FxERC20Child contract on the Mumbai network. This contract will be the child token minted by the root network contract. It should be initialized with the address of the FxPortal contract, the address of the corresponding token on the root network, and the token's name, symbol, and decimals.

4. Deploy an FxERC20RootTunnel contract on the Goerli network. This contract will map tokens from the root network to the child network and send deposits to the child network. When a new token is mapped, the contract deploys a new child token contract using the FxERC20 template and then maps the root token to the child token. The contract sends a message to the child network containing the deposit details when a deposit is made.

   - Constructor arguments:
     - **_checkpointManager** : 0x2890bA17EfE978480615e330ecB65333b880928e
     - **_fxRoot** : 0x3d1d3E34f7fB6D26245E6640E1c50710eFFf15bA
     - **_fxERC20Token** : SafeToken (address of ERC20 token created in step 1)

5. Deploy an FxERC20ChildTunnel contract on the Mumbai network. This contract will be responsible for processing messages from the root network, minting new child tokens, and handling child-to-root withdrawals. When a new token is mapped, the contract deploys a new child token contract using the FxERC20 template and then maps the root token to the child token. When a deposit message is received, the contract mints new child tokens and calls the onTokenTransfer function on the receiver (if it is a contract). When a withdrawal is made, the contract burns the child tokens and sends a message to the root network containing the withdrawal details.

   - Constructor arguments:
     - **_fxChild** : 0xCf73231F28B7331BBe3124B907840A94851f9f11
     - **_tokenTemplate** : FxERC20 address (address of FxERC20 contract created in step 2)

6. On the FxERC20ChildTunnel contract (step 5), set the setFxRootTunnel function. This should be done with the FxERC20RootTunnel contract address from step 4.

7. On the FxERC20RootTunnel contract (step 4), set the setFxChildTunnel function. This should be done with the FxERC20ChildTunnel contract address from step 5.

8. On the SafeToken contract from step 1, call the mint function to mint the desired amount of tokens.

9. On the SafeToken contract, call the approve function with the FxERC20RootTunnel contract address from step 4 and the mint amount from step 9.

10. On the FxERC20RootTunnel contract from step 4, call the deposit function with the SafeToken on the Goerli network from step 1, the user address, the mint amount from step 9, and the data parameter `0x00`.

Additional steps

11. On the FxERC20ChildTunnel contract from step 5, call the balanceOf function to check that the child tokens have been minted.

12. To withdraw the child tokens back to the Goerli network, call the withdraw function on the FxERC20ChildTunnel contract with the amount of child tokens to be burned.
13. Wait for the transaction to be confirmed and the withdrawal to be processed. You can check the status of the withdrawal using the FxExplorer.
14. On the SafeToken contract from step 1, call the balanceOf function to check that the corresponding amount of tokens has been deposited on the Goerli network.

Congratulations, you have successfully tested the FxPortal functionality for transferring tokens between the Goerli and Mumbai networks!

id: matic-to-ethereum title: Transfer data from Polygon to Ethereum description: Transfer state or data from Polygon to Ethereum via Contracts keywords: - docs - matic image: https://matic.network/banners/matic-network-16x9.png

import useBaseUrl from '@docusaurus/useBaseUrl';

Mechanism for transferring data from Polygon to Ethereum is a little different from doing the same for Ethereum to Polygon. The **checkpoint** transactions created by the Validators on the Ethereum chain are used for achieving this. Basically a transaction is initially created on Polygon. While creating this transaction it has to be ensured that an **event is emitted** and the **event logs includes the data we wish to transfer** from Polygon to Ethereum.

In a period of time (about 10-30 mins), this transaction is check-pointed on the Ethereum chain by the validators. Once checkpointing is done, the hash of the transaction created on the Polygon chain can be submitted as a proof on the **RootChainManager** contract on the Ethereum chain. This contract, validates the transaction, verifies that this transaction is included in the checkpoint and finally decodes the event logs from this transaction.

Once this phase is over, we can use the **decoded event log data to perform any change** on the root contract deployed on the Ethereum chain. For this we also need to ensure that, the change of state on Ethereum is only done in a secure way. Hence, we make use of a **Predicate** contract which is a special type of contract that can be only triggered by the **RootChainManager** contract. This architecture ensures that the state changes on Ethereum happens only when the transaction on Polygon is check pointed and verified on the Ethereum chain by the **RootChainManager** contract.

Overview

- A transaction is executed on the child contract deployed on the Polygon chain.
- An event is also emitted in this transaction. The parameters of this **event includes the data which has to be transferred** from Polygon to Ethereum.
- The validators on the Polygon network picks up this transaction in a specific interval of time(probably 10-30mins), validates them and **adds them to the checkpoint** on Ethereum.
- A checkpoint transaction is created on the **RootChain** contract and the checkpoint inclusion can be checked using this script
- Once the checkpoint addition is completed, the **matic.js** library can be used to call the **exit** function of the **RootChainManager** contract. **exit** function can be called using the matic.js library as shown in this example.
- Running the script, verifies the inclusion of the Polygon transaction hash on Ethereum chain, and then in turn calls the **exitToken** function of the predicate contract.
- This ensures that the **state change on the root chain contract** is always done in a **secure** way and **only through the predicate contract**.
- The important thing to note is that the **verification of the transaction hash** from Polygon and **triggering the predicate** contract happens in a **single transaction** and thus ensuring security of any state change on root contract.

Implementation

This is a simple demonstration of how data can be transferred from Polygon to Ethereum. This tutorial shows an example of transferring a uint256 value across the chain. But you can transfer type of data. But it is necessary to encode the data in bytes and then emit it from the child contract. It can be finally decoded at the root contract.

1. First create the root chain and child chain contract. Ensure that the function that does the state change also emits an event. This event must include the data to be transferred as one of its parameters. A sample format of how the Child and Root contract must look like is given below. This is a very simple contract that has a data variable whose value is set by using a setData function. Calling the setData function emits the Data event. Rest of the things in the contract will be explained in the upcoming sections of this tutorial.

A. Child Contract

```
contract Child {

    event Data(address indexed from, bytes bytes_data);

    uint256 public data;

    function setData(bytes memory bytes_data) public {
     data = abi.decode(bytes_data,(uint256));
     emit Data(msg.sender,bytes_data);
    }

}
```

B. Root Contract

Pass this `0x1470E07a6dD1D11eAE439Acaa6971C941C9EF48f` as the value for `_predicate` in the root contract constructor.

```
contract Root {

    address public predicate;
    constructor(address _predicate) public{
        predicate=_predicate;
    }

   modifier onlyPredicate() {
        require(msg.sender == predicate);
        _;
    }

    uint256 public data;

    function setData(bytes memory bytes_data) public onlyPredicate{
        data = abi.decode(bytes_data,(uint256));
    }

}
```

2. Once the child and root contract is deployed on the Polygon and Ethereum chain respectively, these contracts have to be mapped using the PoS bridge. This mapping ensures that a connection is maintained between these two contracts across the chains. For doing this mapping,the Polygon team can be reached on discord.

3. One important thing to note is that, in the root contract, there is a onlyPredicate modifier. It is recommended to use this modifier always because it ensures that only the predicate contract makes the state change on the root contract. The predicate contract is a special contract that triggers the root contract only when the transaction that happened on the Polygon chain is verified by the RootChainManager on Ethereum chain. This ensures secure change of state on the root contract.

For testing the above implementation, we can create a transaction on the Polygon chain by calling the **setData** function of the child contract. We need to wait at this point for the checkpoint to be completed. The checkpoint inclusion can be checked using this script. Once checkpoint is completed, call the exit function of the RootChainManager using the matic.js SDK.

```
const txHash =
  "0xc094de3b7abd29f23a23549d9484e9c6bddb2542e2cc0aa605221cb55548951c";

const logEventSignature =
  "0x93f3e547dcb3ce9c356bb293f12e44f70fc24105d675b782bd639333aab70df7";

const execute = async () => {
  try {
    const tx = await maticPOSClient.posRootChainManager.exit(
      txHash,
      logEventSignature
    );
    console.log(tx.transactionHash); // eslint-disable-line
  } catch (e) {
    console.error(e); // eslint-disable-line
  }
};
```

As shown in the above screenshot, the **txHash** is the transaction hash of the transaction that happened on the child contract deployed on Polygon chain.

The **logEventSignature** is the keccack-256 hash of the Data event. This is the same hash that we have included in the Predicate contract. All the contract code used for this tutorial and the exit script can be found here

Once the exit script is completed, the root contract on Ethereum chain can be queried to verify if the value of the variable data that was set in child contract has also been reflected in the data variable of the root contract.

id: state-transfer title: State Transfer description: Transfer state or data easily from Ethereum to Polygon. keywords: - docs - polygon - polygon wiki - state transfer - ethereum image: https://wiki.polygon.technology/img/polygon-logo.png

Polygon validators continuously monitor a contract on Ethereum chain called `StateSender`. Each time a registered contract on Ethereum chain calls this contract, it emits an event. Using this event Polygon validators relay the data to another contract on Polygon chain. This **State Sync** mechanism is used to send data from Ethereum to Polygon.

Additionally, Polygon validators send an Ethereum hash of each transaction on the Polygon chain on a regular basis. You can use this **checkpoint** to validate any transaction that took place on Polygon. Once a transaction has been verified to have occurred on the Polygon chain, Ethereum can then be used to carry out the appropriate action.

These 2 mechanisms can be used together to enable two-way data (state) transfer between Ethereum and Polygon. To abstract out all these interactions, you can directly inherit our `FxBaseRootTunnel` (on Ethereum) and `FxBaseChildTunnel` (on Polygon) contracts.

Root Tunnel Contract

Use the `FxBaseRootTunnel` contract from here. This contract gives access to the following functions:

- `function _processMessageFromChild(bytes memory data)` : This is a virtual function that needs to be implemented in the contract which inherits it to handle data being sent from `ChildTunnel` .
- `_sendMessageToChild(bytes memory message)` : This function can be called internally with any bytes data as a message. This data will be sent as it is to the child tunnel.
- `receiveMessage(bytes memory inputData)` : This function needs to be called to receive the message emitted by `ChildTunnel` . The proof of transaction needs to be provided as calldata. An example script to generate proof using **matic.js** is included below.

Child Tunnel Contract

Use the `FxBaseChildTunnel` contract from here. This contract gives access to following functions:

- `function _processMessageFromRoot(uint256 stateId, address sender, bytes memory data)` : This is a virtual function that needs to implement the logic to handle messages sent from the `RootTunnel` .
- `function _sendMessageToRoot(bytes memory message)` : This function can be called internally to send any bytes message to the root tunnel.

Prerequisites

- You need to inherit `FxBaseRootTunnel` contract in your root contract on Ethereum. As an example, you can follow this contract . Similarly, inherit `FxBaseChildTunnel` contract in your child on Polygon. Follow this contract as an example.
- While deploying your root contract on
  - **Goerli Testnet**, pass the address of `_checkpointManager` as **0x2890bA17EfE978480615e330ecB65333b880928e** and `_fxRoot` as **0x3d1d3E34f7fB6D26245E6640E1c50710eFFf15bA**.
  - **Ethereum Mainnet**, `_checkpointManager` is **0x86e4dc95c7fbdbf52e33d563bbdb00823894c287** and `_fxRoot` is **0xfe5e5D361b2ad62c541bAb87C45a0B9B018389a2**.
- For deploying the child contract on **Mumbai testnet**, pass **0xCf73231F28B7331BBe3124B907840A94851f9f11** as `_fxChild` in constructor. For **Polygon mainnet,** `_fxChild` will be **0x8397259c983751DAf40400790063935a11afa28a**.
- Call `setFxChildTunnel` on deployed root tunnel with the address of child tunnel. Example: 0x79cd30ace561a226258918b56ce098a08ce0c70707a80bba91197f127a48b5c2
- Call `setFxRootTunnel` on deployed child tunnel with address of root tunnel. Example: 0xffd0cda35a8c3fd6d8c1c04cd79a27b7e5e00cfc2ffc4b864d2b45a8bb7e98b8

Example Contracts of State Transfer Bridge

- **Contracts**: Fx-Portal Github Repository
- **Goerli:** 0xc4432e7dab6c1b43f4dc38ad2a594ca448aec9af
- **Mumbai:** 0xa0060Cc969d760c3FA85844676fB654Bba693C22

State Transfer from Ethereum → Polygon

- You need to call `_sendMessageToChild()` internally in your root contract and pass the data as an argument to be sent to Polygon. Example: 0x28705fcae757a0c88694bd167cb94a2696a0bc9a645eb4ae20cff960537644c1
- In your child contract, implement `_processMessageFromRoot()` virtual function in `FxBaseChildTunnel` to retrieve data from Ethereum. The data will be received automatically from the state receiver when the state is synced.

State Transfer from Polygon → Ethereum

1. Call `_sendMessageToRoot()` internally in your child contract with data as a parameter to be sent to Ethereum. Example: 0x3cc9f7e675bb4f6af87ee99947bf24c38cbffa0b933d8c981644a2f2b550e66a

Note the transaction hash as it will be used to generate proof after it has been included as a checkpoint.

2. **Proof Generation to complete the exit on root chain**: Generate the proof using the **tx hash** and **MESSAGE_SENT_EVENT_SIG**. To generate the proof, you can either use the proof generation API hosted by Polygon or you can also spin up your own proof generation API by following the instructions here.

The proof generation endpoint hosted by Polygon is available here.

```
- `burnTxHash` is the transaction hash of the `_sendMessageToRoot()` transaction you initiated on Polygon.
- `eventSignature` is the event signature of the event emitted by the `_sendMessageToRoot()` function. The event signature for the
MESSAGE_SENT_EVENT_SIG is `0x8c5261668696ce22758910d05bab8f186d6eb247ceac2af2e82c7dc17669b036`.
```

The proof generation API usage examples for the Mainnet and Testnet are as follows:-

→ Mumbai Testnet Proof generation

→ Polygon Mainnet Proof generation

3. Implement `_processMessageFromChild()` in your root contract.

4. Use the generated proof as an input to `receiveMessage()` to retrieve data sent from child tunnel into your contract. Example: 0x436dcd500659bea715a09d9257295ddc21290769daeea7f0b666166ef75e3515)

---

id: how-state-sync-works title: How does State Sync work? description: "Sending the state from the Ethereum chain to the Bor chain." keywords: - docs - matic - state sync - working image: https://matic.network/banners/matic-network-16x9.png

How does State Sync work?

State management sends the state from the Ethereum chain to the Bor chain. It is called **state-sync**.

State transfer from Ethereum to Bor happens through system call. Suppose, a user deposits USDC to the deposit manager on Ethereum. Validators listen to those events, validate, and store them in Heimdall state. Bor gets the latest state-sync records and updates the Bor state (mints equal amount of USDC on Bor) using a system call.

State sender

Source: https://github.com/maticnetwork/contracts/blob/develop/contracts/root/stateSyncer/StateSender.sol

To sync state, the contract calls following method **state sender contract** on Ethereum chain.

```
contract StateSender {
    /**
     * Emits `stateSynced` events to start sync process on Ethereum chain
     * @param receiver    Target contract on Bor chain
     * @param data        Data to send
     */
    function syncState (
        address receiver,
        bytes calldata data
    ) external;
}
```

`receiver` contract must be present on the child chain, which receives state `data` once the process is complete. `syncState` emits `StateSynced` event on Ethereum, which is the following:

```
/**
 * Emits `stateSynced` events to start sync process on Ethereum chain
 * @param id                State id
 * @param contractAddress   Target contract address on Bor
 * @param data              Data to send to Bor chain for Target contract address
 */
event StateSynced (
    uint256 indexed id,
    address indexed contractAddress,
    bytes data
);
```

Once the `StateSynced` event emitted on the `stateSender` contract on the Ethereum chain, Heimdall listens to those events and adds to the Heimdall state after 2/3+ validators agree on the.

After every sprint (currently 64 blocks on Bor), Bor fetches new state-sync record and updates the state using a `system` call. Here is the code for the same: https://github.com/maticnetwork/bor/blob/6f0f08daecaebbff44cf18bee558fc3796d41832/consensus/bor/genesis_contracts_client.go#L51

During `commitState`, Bor executes `onStateReceive`, with `stateId` and `data` as args, on target contract.

State receiver interface on Bor

`receiver` contract on Bor chain must implement following interface.

```
// IStateReceiver represents interface to receive state
interface IStateReceiver {
  function onStateReceive(uint256 stateId, bytes calldata data) external;
}
```

Only `0x0000000000000000000000000000000000001001` — `StateReceiver.sol`, must be allowed to call `onStateReceive` function on target contract.

System Call

Only system address, `2^160-2`, allows making a system call. Bor calls it internally with the system address as `msg.sender`. It changes the contract state and updates the state root for a particular block. Inspired from https://github.com/ethereum/EIPs/blob/master/EIPS/eip-210.md and https://wiki.parity.io/Validator-Set#contracts

System call is helpful to change state to contract without making any transaction.

State-sync logs and Bor Block Receipt

Events emitted by system calls are handled in a different way than normal logs. Here is the code: https://github.com/maticnetwork/bor/pull/90.

Bor produces a new tx / receipt just for the client which includes all the logs for state-sync. Tx hash is derived from block number and block hash (last block at that sprint):

```
keccak256("matic-bor-receipt-" + block number + block hash)
```

This doesn't change any consensus logic, only client changes. `eth_getBlockByNumber`, `eth_getTransactionReceipt`, and `eth_getLogs` include state-sync logs with derived. Note that the bloom filter on the block doesn't include inclusion for state-sync logs. It also doesn't include derived tx in `transactionRoot` or `receiptRoot`.--- id: state-sync title: State Sync Mechanism description: Mechanism to read Ethereum data from Matic EVM chain keywords: - docs - matic - state sync - mechanism - read ethereum data image: https://matic.network/banners/matic-network-16x9.png

State Sync Mechanism

`State Sync` is the native mechanism to read Ethereum data from Matic EVM chain.

Validators on the Heimdall layer pickup the StateSynced event and pass it on to the Bor layer (read more about the architecture here.

The receiver contract inherits IStateReceiver, and custom logic sits inside onStateReceive function.

This is the flow required from dapps / users to work with state-sync:

1. Call the smart contract function present here: https://github.com/maticnetwork/contracts/blob/19163ddecf91db17333859ae72dd73c91bee6191/contracts/root/stateSyncer/StateSender.sol#L33
2. Which emits an event called `StateSynced(uint256 indexed id, address indexed contractAddress, bytes data);`
3. All the validators on the Heimdall chain receive this event and one of them, whoever wishes to get the tx fees for state sync sends this transaction to Heimdall.
4. Once `state-sync` transaction on Heimdall has been included in a block, it is added to pending state-sync list.
5. After every sprint on `bor`, the Bor node fetches the pending state-sync events from Heimdall via an API call.
6. The receiver contract inherits `IStateReceiver` interface, and custom logic of decoding the data bytes and performing any action sits inside `onStateReceive` function: https://github.com/maticnetwork/genesis-contracts/blob/master/contracts/IStateReceiver.sol

---

id: antehandler title: Ante Handler description: Ante Handler checks and validates the transaction keywords: - docs - matic - polygon - Ante Handler - validate transactions image: https://matic.network/banners/matic-network-16x9.png

Ante Handler

Ante handler checks and validates the transaction. After the verification, it checks the balance of the sender for enough fees and deduct fees in case of successful transaction inclusion.

Gas Limit

Each block and transaction have a limit for gas usage. A block can contain multiple transactions, but gas used by all transactions in a block must be less than block gas limit to avoid larger blocks.

```
block.GasLimit >= sum(tx1.GasUsed + tx2.GasUsed + ..... + txN.GasUsed)
```

Note that each state manipulation on transaction costs gas, including signature verification for the transaction.

Block Gas Limit

Max block gas limit and bytes per block is passed while setting up app's consensus params: https://github.com/maticnetwork/heimdall/blob/develop/app/app.go#L464-L471

```
maxGasPerBlock   int64 = 10000000 // 10 Million
maxBytesPerBlock int64 = 22020096 // 21 MB

// pass consensus params
ConsensusParams: &abci.ConsensusParams{
    Block: &abci.BlockParams{
        MaxBytes: maxBytesPerBlock,
        MaxGas:   maxGasPerBlock,
    },
    ...
},
```

Transaction Gas Limit

The transaction gas limit is defined in params in `auth` module. It can be changed through the Heimdall `gov` module.

Checkpoint Transaction Gas Limit

Since block contains multiple transactions and verifies this particular transaction on the Ethereum chain, Merkle proof is required. To avoid extra Merkle proof verification for checkpoint transaction, Heimdall only allows one transaction in the block if the transaction type is `MsgCheckpoint`

```
// fee wanted for checkpoint transaction
gasWantedPerCheckpoinTx sdk.Gas = 10000000 // 10 Million

// checkpoint gas limit
if stdTx.Msg.Type() == "checkpoint" && stdTx.Msg.Route() == "checkpoint" {
    gasForTx = gasWantedPerCheckpoinTx
}
```

Transaction Verification and Replay Protection

Ante Handler handles and verifies signature in incoming transaction: https://github.com/maticnetwork/heimdall/blob/develop/auth/ante.go#L230-L266

Each transaction must include `sequenceNumber` to avoid replay attacks. After each successful transaction inclusion, Ante handler increases the sequence number for the TX sender account to avoid duplication (replay) of the previous transactions.--- id: checkpoint title: Checkpoint description: Snapshots of the Bor chain state submitted to Ethereum keywords: - docs - matic - polygon - checkpoint - snapshots of bor chain - ethereum image: https://matic.network/banners/matic-network-16x9.png

import useBaseUrl from '@docusaurus/useBaseUrl';

Checkpoint

Checkpoints are the most crucial part of the Polygon network. It represents snapshots of the Bor chain state and is supposed to be attested by ⅔+ of the validator set before it is validated and submitted on the contracts deployed on Ethereum.

Types

Checkpoint structure on Heimdall state looks like following:

```
type CheckpointBlockHeader struct {
    // Proposer is selected based on stake
    Proposer        types.HeimdallAddress `json:"proposer"`

    // StartBlock: The block number on Bor from which this checkpoint starts
    StartBlock      uint64                `json:"startBlock"`

    // EndBlock: The block number on Bor from which this checkpoint ends
    EndBlock        uint64                `json:"endBlock"`

    // RootHash is the Merkle root of all the leaves containing the block
    // headers starting from start to the end block
    RootHash        types.HeimdallHash    `json:"rootHash"`

    // Account root hash for each validator
  // Hash of data that needs to be passed from Heimdall to Ethereum chain like withdraw topup etc.
    AccountRootHash types.HeimdallHash    `json:"accountRootHash"`

  // Timestamp when checkpoint was created on Heimdall
    TimeStamp       uint64          `json:"timestamp"`
}
```

Root hash

<img src={useBaseUrl("img/checkpoint/checkpoint.svg")} />

`RootHash` is the Merkle hash of Bor block hashes from `StartBlock` to `EndBlock`. Root hash for the checkpoint is created using the following way:

```
blockHash = keccak256([number, time, tx hash, receipt hash])
```

Pseudocode for the root hash for `1` to `n` Bor blocks:

```
B(1) := keccak256([number, time, tx hash, receipt hash])
B(2) := keccak256([number, time, tx hash, receipt hash])
.
```

```
.
.
B(n) := keccak256([number, time, tx hash, receipt hash])

// checkpoint is Merkle root of all block hash
checkpoint's root hash = Merkel[B(1), B(2), ....., B(n)]
```

Here are some snippets of how checkpoint is created from Bor chain block headers.

Source: https://github.com/maticnetwork/heimdall/blob/develop/checkpoint/types/merkel.go#L60-L114

```
// Golang representation of block data used in checkpoint
blockData := crypto.Keccak256(appendBytes32(
    blockHeader.Number.Bytes(),
    new(big.Int).SetUint64(blockHeader.Time).Bytes(),
    blockHeader.TxHash.Bytes(),
    blockHeader.ReceiptHash.Bytes(),
))

// array of block hashes of Bor blocks
headers := [blockData1, blockData2, ..., blockDataN]

// merkel tree
tree := merkle.NewTreeWithOpts(merkle.TreeOptions{EnableHashSorting: false, DisableHashLeaves: true})
tree.Generate(convert(headers), sha3.NewLegacyKeccak256())

// create checkpoint's root hash
rootHash := tree.Root().Hash
```

AccountRootHash

`AccountRootHash` is the hash of the validator account-related information that needs to pass to the Ethereum chain at each checkpoint.

```
eachAccountHash := keccak256([validator id, withdraw fee, slash amount])
```

Pseudocode for the account root hash for `1` to `n` Bor blocks:

```
B(1) := keccak256([validator id, withdraw fee, slash amount])
B(2) := keccak256([validator id, withdraw fee, slash amount])
.
.
.
B(n) := keccak256([validator id, withdraw fee, slash amount])

// account root hash is Merkle root of all block hash
checkpoint's account root hash = Merkel[B(1), B(2), ....., B(n)]
```

Golang code for the account hash can be found here: https://github.com/maticnetwork/heimdall/blob/develop/types/dividend-account.go#L91-L101

```
// DividendAccount contains Fee, Slashed amount
type DividendAccount struct {
    ID            DividendAccountID `json:"ID"`
    FeeAmount     string            `json:"feeAmount"`     // string representation of big.Int
    SlashedAmount string            `json:"slashedAmount"` // string representation of big.Int
}

// calculate hash for particular account
func (da DividendAccount) CalculateHash() ([]byte, error) {
    fee, _ := big.NewInt(0).SetString(da.FeeAmount, 10)
    slashAmount, _ := big.NewInt(0).SetString(da.SlashedAmount, 10)
    divAccountHash := crypto.Keccak256(appendBytes32(
        new(big.Int).SetUint64(uint64(da.ID)).Bytes(),
        fee.Bytes(),
        slashAmount.Bytes(),
    ))

    return divAccountHash, nil
}
```

---

id: encoder title: Encoder (Pulp) description: RLP encoding to produce special transactions, like checkpoint keywords: - docs - matic - rlp encoding - checkpoint - encoder - polygon image: https://matic.network/banners/matic-network-16x9.png

Encoder (Pulp)

Heimdall needs to verify the transactions of Heimdall on the Ethereum chain. For that it uses RLP encoding to produce special transactions, like checkpoint.

This special transaction uses `pulp` (RLP based) encoding instead of default amino encoding.

Pulp uses a prefix-based simple encoding mechanism to solve interface decoding. Check `GetPulpHash` method.

Source: https://github.com/maticnetwork/heimdall/blob/master/auth/types/pulp.go

```go
const (
    // PulpHashLength pulp hash length
    PulpHashLength int = 4
)

// GetPulpHash returns string hash
func GetPulpHash(name string) []byte {
    return crypto.Keccak256([]byte(name))[:PulpHashLength]
}
```

The below returns prefix-bytes for a given `msg`. Here is an example on how to register an object for pulp encoding:

```go
RegisterConcrete(name, obj) {
    rtype := reflect.TypeOf(obj)
    // set record for name => type of the object
    p.typeInfos[hex.EncodeToString(GetPulpHash(name))] = rtype
}

// register "A"
pulp.RegisterConcrete("A", A{})
```

Encoding is just RLP encoding and prepending hash of `GetPulpHash` of the `name`:

```go
// EncodeToBytes encodes msg to bytes
txBytes, err := rlp.EncodeToBytes(obj)
if err != nil {
    return nil, err
}

result := append(GetPulpHash("A"), txBytes[:]...), nil
```

Decoding works as follows:

```go
// retrieve type of object based on prefix
rtype := typeInfos[hex.EncodeToString(incomingData[:PulpHashLength])]

// create new object
newMsg := reflect.New(rtype).Interface()

// decode without prefix and inject into newly created object
if err := rlp.DecodeBytes(incomingData[PulpHashLength:], newMsg); err != nil {
    return nil, err
}

// result => newMsg
```

:::info For more information

The Cosmos SDK utilizes two binary wire encoding protocols, Amino and Protocol Buffers, where Amino is an object encoding specification. It is a subset of Proto3 with an extension for interface support. See the Proto3 spec for more information on Proto3, which Amino is largely compatible with (but not with Proto2).

More here: https://docs.cosmos.network/master/core/encoding.html

:::

id: auth title: Auth description: Module for specifying base transaction and account types keywords: - docs - matic - auth module - transaction - account types image: https://matic.network/banners/matic-network-16x9.png

Auth Module

This document describes the `auth` module of Heimdall.

The `auth` module is responsible for specifying the base transaction and account types for an application. It contains the ante handler, where all basic transaction validity checks (signatures, nonces, auxiliary fields) are performed, and exposes the account keeper, which allows other modules to read, write, and modify accounts.

Gas and Fees

Fees serve two purposes for an operator of the network.

Fees limit the growth of the state stored by every full node and allow for general purpose censorship of transactions of little economic value. Fees are best suited as an anti-spam mechanism where validators are disinterested in the use of the network and identities of users.

Since Heimdall doesn't support custom contract or code for any transaction, it uses fixed cost transactions. For fixed cost transactions, the validator can top up their accounts on the Ethereum chain and get tokens on Heimdall using the Topup module.

Types

Besides accounts (specified in State), the types exposed by the auth module are **StdSignature**, the combination of an optional public key and a cryptographic signature as a byte array, **StdTx**, a struct that implements the `sdk.Tx` interface using **StdSignature**, and **StdSignDoc**, a replay-prevention structure for **StdTx** which transaction senders must sign over.

StdSignature

A `StdSignature` is the types of a byte array.

```
// StdSignature represents a sig
type StdSignature []byte
```

StdTx

A `StdTx` is a struct that implements the `sdk.Tx` interface, and is likely to be generic enough to serve the purposes of many types of transactions.

```
type StdTx struct {
        Msg       sdk.Msg      `json:"msg" yaml:"msg"`
        Signature StdSignature `json:"signature" yaml:"signature"`
        Memo      string       `json:"memo" yaml:"memo"`
}
```

StdSignDoc

A `StdSignDoc` is a replay-prevention structure to be signed over, which ensures that any submitted transaction (which is simply a signature over a particular byte string) will only be executable once on a Heimdall.

```
// StdSignDoc is replay-prevention structure.
// It includes the result of msg.GetSignBytes(),
// as well as the ChainID (prevent cross chain replay)
// and the Sequence numbers for each signature (prevent
// inchain replay and enforce tx ordering per account).
type StdSignDoc struct {
    ChainID       string          `json:"chain_id" yaml:"chain_id"`
    AccountNumber uint64          `json:"account_number" yaml:"account_number"`
    Sequence      uint64          `json:"sequence" yaml:"sequence"`
    Msg           json.RawMessage `json:"msg" yaml:"msg"`
    Memo          string          `json:"memo" yaml:"memo"`
}
```

Account

It manages addresses, coins and nonce for transactions. It also signs and validates transactions.

Source: https://github.com/maticnetwork/heimdall/blob/master/auth/types/account.go#L32-L54

```
type BaseAccount struct {
        Address types.HeimdallAddress `json:"address" yaml:"address"`
        Coins types.Coins `json:"coins" yaml:"coins"`
        PubKey crypto.PubKey `json:"public_key" yaml:"public_key"`
        AccountNumber uint64 `json:"account_number" yaml:"account_number"`
        Sequence uint64 `json:"sequence" yaml:"sequence"`
}
```

Parameters

The auth module contains the following parameters:

Key	Type	Default value
MaxMemoCharacters	uint64	256

TxSigLimit	uint64	7
TxSizeCostPerByte	uint64	10
SigVerifyCostED25519	uint64	590
SigVerifyCostSecp256k1	uint64	1000
DefaultMaxTxGas	uint64	1000000
DefaultTxFees	string	"1000000000000000"

CLI Commands

Show account

To print account related data into Heimdall;

```
heimdalld show-account
```

Expected Result:

```
{
    "address": "0x68243159a498cf20d945cf3E4250918278BA538E",
    "pub_key":
"0x040a9f6879c7cdab7ecc67e157cda15e8b2ddbde107a04bc22d02f50032e393f6360a05e85c7c1ecd201ad30dfb886af12dd02b47e4463f6f0f6f94159dc9f1(
}
```

Account and coin details

To display account details, coins, sequence and account number;

```
heimdallcli query auth account 0x68243159a498cf20d945cf3E4250918278BA538E --trust-node
```

Expected Result:

```
address: 0x68243159a498cf20d945cf3e4250918278ba538e
coins:
- denom: matic
    amount:
    i: "10000000000000000000"
pubkey: ""
accountnumber: 0
sequence: 0
```

Parameters

To print all params;

```
heimdallcli query auth params
```

Expected Result:

```
max_memo_characters: 256
tx_sig_limit: 7
tx_size_cost_per_byte: 10
sig_verify_cost_ed25519: 590
sig_verify_cost_secp256k1: 1000
max_tx_gas: 1000000
tx_fees: "1000000000000000"
```

REST APIs

Name	Endpoint	Description
Account details	/auth/accounts/{address}	Returns all details for an address
Account sequence details	/auth/accounts/{address}/sequence	Returns only necessary details for signing
Auth params	/auth/params	Returns all params auth module uses

---

id: bank title: Bank description: Module handling account balance transfers for Heimdall keywords: - docs - matic - bank - account balance - heimdall image: https://matic.network/banners/matic-network-16x9.png

Bank Module

The `bank` module handles account balance transfers for Heimdall. This module corresponds to the `bank` module from cosmos-sdk.

Messages

MsgSend

`MsgSend` handles transfer between accounts in Heimdall. Here is a structure for transaction message:

```
// MsgSend - high-level transaction of the coin module
type MsgSend struct {
    FromAddress types.HeimdallAddress `json:"from_address"`
    ToAddress   types.HeimdallAddress `json:"to_address"`
    Amount      types.Coins           `json:"amount"`
}
```

MsgMultiSend

`MsgMultiSend` handles multi transfer between account for Heimdall.

```
// MsgMultiSend - high-level transaction of the coin module
type MsgMultiSend struct {
    Inputs  []Input  `json:"inputs"`
    Outputs []Output `json:"outputs"`
}
```

Parameters

The bank module contains the following parameters:

Key	Type	Default value
`sendenabled`	bool	true

CLI Commands

Send Balance

Following command will send 1000 matic tokens to mentioned `address`;

```
heimdallcli tx bank send <address> 1000matic --chain-id <chain-id>
```---
id: bor
title: Bor
description: Module that handles span management on Heimdall
keywords:
  - docs
  - matic
  - bor module
  - heimdall
image: https://matic.network/banners/matic-network-16x9.png

Bor Module

Bor module handles span management on Heimdall. Given Bor chain's current block number `n`, current span `span`, if
`span.StartBlock <= n < span.EndBlock`, new span is proposed on Heimdall by any validator.

Messages

MsgProposeSpan

`MsgProposeSpan` sets the validators' committee for a given `span` and stores a new span into Heimdall state.

Source: [https://github.com/maticnetwork/heimdall/blob/develop/bor/handler.go#L27]
(https://github.com/maticnetwork/heimdall/blob/develop/bor/handler.go#L27)

```go
// MsgProposeSpan creates msg propose span
type MsgProposeSpan struct {
    ID         uint64                 `json:"span_id"`
    Proposer   hmTypes.HeimdallAddress `json:"proposer"`
    StartBlock uint64                 `json:"start_block"`
    EndBlock   uint64                 `json:"end_block"`
```

```
    ChainID     string                  `json:"bor_chain_id"`
}
```

Here is how this transaction chooses producers out of all validators:

1. It creates multiple slots based on validators' power. Example: A with power 10 will have 10 slots, B with power 20 with have 20 slots.
2. With all slots, `shuffle` function shuffles them using `seed` and selects first `producerCount` producers. `bor` module on Heimdall uses ETH 2.0 shuffle algorithm to choose producers out of all validators. Each span `n` uses block hash of Ethereum (ETH 1.0) block `n` as `seed`. Note that slots based selection allows validators to get selected based on their power. The higher power validator will have a higher probability to get selected. Source: https://github.com/maticnetwork/heimdall/blob/develop/bor/selection.go

```
// SelectNextProducers selects producers for the next span by converting power to slots
// spanEligibleVals - all validators eligible for next span
func SelectNextProducers(blkHash common.Hash, spanEligibleVals []hmTypes.Validator, producerCount uint64) (selectedIDs []uint64,
err error) {
    if len(spanEligibleVals) <= int(producerCount) {
        for _, val := range spanEligibleVals {
            selectedIDs = append(selectedIDs, uint64(val.ID))
        }
        return
    }

    // extract seed from hash
    seed := helper.ToBytes32(blkHash.Bytes()[:32])
    validatorIndices := convertToSlots(spanEligibleVals)
    selectedIDs, err = ShuffleList(validatorIndices, seed)
    if err != nil {
        return
    }
    return selectedIDs[:producerCount], nil
}

// converts validator power to slots
func convertToSlots(vals []hmTypes.Validator) (validatorIndices []uint64) {
    for _, val := range vals {
        for val.VotingPower >= types.SlotCost {
            validatorIndices = append(validatorIndices, uint64(val.ID))
            val.VotingPower = val.VotingPower - types.SlotCost
        }
    }
    return validatorIndices
}
```

Types

Here are the span details that Heimdall uses:

```
// Span structure
type Span struct {
    ID               uint64       `json:"span_id" yaml:"span_id"`
    StartBlock       uint64       `json:"start_block" yaml:"start_block"`
    EndBlock         uint64       `json:"end_block" yaml:"end_block"`
    ValidatorSet     ValidatorSet `json:"validator_set" yaml:"validator_set"`
    SelectedProducers []Validator  `json:"selected_producers" yaml:"selected_producers"`
    ChainID          string       `json:"bor_chain_id" yaml:"bor_chain_id"`
}
```

Parameters

The Bor module contains the following parameters:

Key	Type	Default value	Duration (*)
SprintDuration.	uint64	16 blocks	32 seconds
SpanDuration	uint64	100 * SprintDuration = 1,600 blocks	3,200 seconds (53min and 20s)
ProducerCount	uint64	4 blocks	8 seconds

(*): Given that blocks are produced every 2 seconds on Bor.

Previously, a sprint would last 64 blocks but it was agreed to decrease this number to 16 blocks on the Delhi hard fork of January 17th, 2023, precisely starting at block number 38,189,056.

CLI Commands

Span propose tx

```
heimdallcli tx bor propose-span \
    --start-block <start-block> \
    --chain-id <heimdall-chain-id>
```

Query current span

```
heimdallcli query bor span latest-span --chain-id <heimdall-chain-id>
```

Expected output:


```
{
  "span_id":2,
  "start_block":6656,
  "end_block":13055,
  "validator_set":{
    "validators":[
      {
        "ID":1,
        "startEpoch":0,
        "endEpoch":0,
        "power":1,

"pubKey":"0x04b12d8b2f6e3d45a7ace12c4b2158f79b95e4c28ebe5ad54c439be9431d7fc9dc1164210bf6a5c3b8523528b931e772c86a307e8cff4b725e6b4a7
        "signer":"0x6c468cf8c9879006e22ec4029696e005c2319c9d",
        "last_updated":"",
        "accum":0
      }
    ],
    "proposer":{
      "ID":1,
      "startEpoch":0,
      "endEpoch":0,
      "power":1,

"pubKey":"0x04b12d8b2f6e3d45a7ace12c4b2158f79b95e4c28ebe5ad54c439be9431d7fc9dc1164210bf6a5c3b8523528b931e772c86a307e8cff4b725e6b4a7
      "signer":"0x6c468cf8c9879006e22ec4029696e005c2319c9d",
      "last_updated":"",
      "accum":0
    }
  },
  "selected_producers":[
    {
      "ID":1,
      "startEpoch":0,
      "endEpoch":0,
      "power":1,

"pubKey":"0x04b12d8b2f6e3d45a7ace12c4b2158f79b95e4c28ebe5ad54c439be9431d7fc9dc1164210bf6a5c3b8523528b931e772c86a307e8cff4b725e6b4a7
      "signer":"0x6c468cf8c9879006e22ec4029696e005c2319c9d",
      "last_updated":"",
      "accum":0
    }
  ],
  "bor_chain_id":"15001"
}
```


Query span by id

```
heimdallcli query bor span --span-id <span-id> --chain-id <heimdall-chain-id>
```

It prints the result in same format as above.

Parameters

To print all params;

```
heimdalldcli query bor params
```

Expected Result:

```
sprint_duration: 16
span_duration: 1600
producer_count: 4
```

REST APIs

Name	Method	Endpoint
Span details	GET	/bor/span/<span-id>
Get latest span	GET	/bor/latest-span
Get params	GET	/bor/params

---

id: clerk title: Clerk description: Module that manages generic state-sync from Ethereum to Bor keywords: - docs - matic - module - state sync - clerk - heimdall image: https://matic.network/banners/matic-network-16x9.png

Clerk

Clerk manages generic state-sync from Ethereum chain to Bor chain. Heimdall agrees on state sync, which is initiated on the Ethereum chain using the Clerk module.

More details are available in state sync mechanism

Messages

MsgEventRecord

`MsgEventRecord` transaction is responsible for validating events from `StateSender.sol` and storing the state on Heimdall for Bor to use.

Handler for this transaction validates for any given `msg.TxHash` and `msg.LogIndex`. It throws `Older invalid tx found` error if trying to process the transaction more than once.

Here is the structure for the transaction message:

```go
// MsgEventRecord - state msg
type MsgEventRecord struct {
    From     types.HeimdallAddress `json:"from"`
    TxHash   types.HeimdallHash    `json:"tx_hash"`
    LogIndex uint64                `json:"log_index"`
    ID       uint64                `json:"id"`
    ChainID  string                `json:"bor_chain_id"`
}
```

CLI Commands

Send state record transaction

```
heimdallcli tx clerk record
    --log-index <log-index>
    --tx-hash <transaction-hash>
    --bor-chain-id <bor-chain-id>
    --chain-id <heimdall-chain-id>
```

To query already validated state event record

```
heimdallcli query clerk record --id <state-record-id>
```

REST APIs

Name	Method	Endpoint
Event record details	GET	/clerk/event-record/<record-id>
All event records	GET	/clerk/event-record/list

---

id: topup title: Topup description: An amount which will be used to pay fees on Heimdall chain keywords: - docs - matic - topup - fees - heimdall image: https://matic.network/banners/matic-network-16x9.png

Topup

Heimdall Topup is an amount which will be used to pay fees on Heimdall chain.

There are two ways to topup your account:

1. When new validator joins, they can mention a `topup` amount as top-up in addition to the staked amount, which will be moved as balance on Heimdall chain to pays fees on Heimdall.
2. A user can directly call the top-up function on the staking smart contract on Ethereum to increase top-up balance on Heimdall.

Messages

MsgTopup

`MsgTopup` transaction is responsible for minting balance to an address on Heimdall based on Ethereum chain's `TopUpEvent` on staking manager contract.

Handler for this transaction processes top-up and increases the balance only once for any given `msg.TxHash` and `msg.LogIndex`. It throws `Older invalid tx found` error, if trying to process the top-up more than once.

Here is the structure for the top-up transaction message:

```
type MsgTopup struct {
    FromAddress types.HeimdallAddress `json:"from_address"`
    ID          types.ValidatorID     `json:"id"`
    TxHash      types.HeimdallHash    `json:"tx_hash"`
    LogIndex    uint64                `json:"log_index"`
}
```

MsgWithdrawFee

`MsgWithdrawFee` transaction is responsible for withdrawing balance from Heimdall to Ethereum chain. A Validator can withdraw any amount from Heimdall.

Handler processes the withdraw by deducting the balance from the given validator and prepares the state to send the next checkpoint. The next possible checkpoint will contain the withdraw related state for the specific validator.

Handler gets validator information based on `ValidatorAddress` and processes the withdraw.

```
// MsgWithdrawFee - high-level transaction of the fee coin withdrawal module
type MsgWithdrawFee struct {
    ValidatorAddress types.HeimdallAddress `json:"from_address"`
    Amount           types.Int             `json:"amount"`
}
```

CLI Commands

Topup fee

```
heimdallcli tx topup fee
    --log-index <log-index>
    --tx-hash <transaction-hash>
    --validator-id <validator ID here>
    --chain-id <heimdall-chain-id>
```

Withdraw fee

```
heimdallcli tx topup withdraw --chain-id <heimdall-chain-id>
```

To check reflected topup on account run following command

```
heimdallcli query auth account <validator-address> --trust-node
```

REST APIs

Name	Method	URL	Body Params
Topup Fee	POST	/topup/fee	`id` Validator id, `tx_hash` Transaction hash of successful topup event on Ethereum chain, `log_index` Log index of topup event emitted on Ethereum chain
Withdraw Fee	POST	/topup/withdraw	`amount` Withdraw amount

id: chainmanager title: Chain Manager description: Module that provides all necessary dependencies keywords: - docs - matic - chain manager - module - heimdall image: https://matic.network/banners/matic-network-16x9.png

Chain Manager

This document specifies an overview of the chain manager module of Heimdall.

The **chain manager** module provides all necessary dependencies like `contract-addresses`, `bor_chain_id`, and `tx_confirmation_time`. Other parameters can be added to this later on.

Params are updated through the `gov` module.

Types

Chainmanager structure on Heimdall looks like the following:

```
type ChainParams struct {
    // BorChainID is valid bor chainId
    BorChainID            string                  `json:"bor_chain_id" yaml:"bor_chain_id"`

    // MaticTokenAddress is valid matic token address
    MaticTokenAddress     hmTypes.HeimdallAddress `json:"matic_token_address" yaml:"matic_token_address"`

    // StakingManagerAddress is valid contract address
    StakingManagerAddress hmTypes.HeimdallAddress `json:"staking_manager_address" yaml:"staking_manager_address"`

    // RootChainAddress is valid contract address
    RootChainAddress      hmTypes.HeimdallAddress `json:"root_chain_address" yaml:"root_chain_address"`

    // StakingInfoAddress is valid contract address
    StakingInfoAddress    hmTypes.HeimdallAddress `json:"staking_info_address" yaml:"staking_info_address"`

    // StateSendedAddress is valid contract address
    StateSenderAddress    hmTypes.HeimdallAddress `json:"state_sender_address" yaml:"state_sender_address"`

    // Bor Chain Contracts
    // StateReceiveAddress is valid contract address
    StateReceiverAddress hmTypes.HeimdallAddress `json:"state_receiver_address" yaml:"state_receiver_address"`

    // ValidatorSetAddress is valid contract address
    ValidatorSetAddress  hmTypes.HeimdallAddress `json:"validator_set_address" yaml:"validator_set_address"`
}
```

CLI Commands

Parameters

To print all params;

```
heimdallcli query chainmanager params --trust-node
```

Expected Result

```
tx_confirmation_time: 12s
chain_params:
  bor_chain_id: "15001"
  matic_token_address: "0x0000000000000000000000000000000000000000"
  staking_manager_address: "0x00000000000000000000000000000000000000000"
  root_chain_address: "0x0000000000000000000000000000000000000000"
  staking_info_address: "0x0000000000000000000000000000000000000000"
  state_sender_address: "0x0000000000000000000000000000000000000000"
  state_receiver_address: "0x00000000000000000000000000000000000000000"
  validator_set_address: "0x0000000000000000000000000000000000000000"
```

REST APIs

Name	Method	URL
Params	GET	chainmanager/params

All query APIs will provide response in the following format:

```
{
    "height": "1",
    "result": {
        ...
    }
}
```---
id: checkpoint
title: Checkpoint
description: Module that manages checkpoint-related functionalities
keywords:
  - docs
  - matic
  - checkpoint
  - heimdall
image: https://matic.network/banners/matic-network-16x9.png
---

import useBaseUrl from '@docusaurus/useBaseUrl';

Checkpoint

`checkpoint` module manages checkpoint related functionalities for Heimdall. It needs Bor chain when a new checkpoint is proposed on Heimdall to verify checkpoint root hash.

All related to checkpoint data is explained in details [here](/docs/pos/design/heimdall/checkpoint).

Checkpoint life-cycle

Heimdall uses the same leader selection algorithm as Tendermint to select the next proposer. While submitting checkpoints on the Ethereum chain, it may fail due to multiple reasons like gas limit, traffic on Ethereum, high gas fees. That's why the multi-stage checkpoint process is required.

Each checkpoint has validator as proposer. If checkpoint on Ethereum chain fails or succeeds, `ack` and `no-ack` transaction would change the proposer on Heimdall for next checkpoint. The following flow chart represents the life cycle of the checkpoint:

<img src={useBaseUrl("img/checkpoint/checkpoint-flowchart.svg")} />

Messages

<img src={useBaseUrl("img/checkpoint/checkpoint-module-flow.svg")} />

MsgCheckpoint

`MsgCheckpoint` handles checkpoint verification on Heimdall. Only this message uses RLP encoding as it needs to be verified on Ethereum chain.

```go
// MsgCheckpoint represents checkpoint transaction
type MsgCheckpoint struct {
    Proposer        types.HeimdallAddress `json:"proposer"`
    StartBlock      uint64                `json:"startBlock"`
    EndBlock        uint64                `json:"endBlock"`
    RootHash        types.HeimdallHash    `json:"rootHash"`
    AccountRootHash types.HeimdallHash    `json:"accountRootHash"`
}
```

Once this transaction gets processed on Heimdall, the `proposer` takes `votes` and `sigs` from Tendermint for this transaction and sends checkpoint on the Ethereum chain.

Since block contains multiple transactions and verifies this particular transaction on the Ethereum chain, Merkle proof is required. To avoid extra Merkle proof verification on Ethereum, Heimdall only allows one transaction in the block if the transaction type is `MsgCheckpoint`

To allow this mechanism, Heimdall sets `MsgCheckpoint` transaction as high gas consumed transaction. Check https://github.com/maticnetwork/heimdall/blob/develop/auth/ante.go#L104-L106

```
// fee wanted for checkpoint transaction
gasWantedPerCheckpoinTx sdk.Gas = 10000000

// checkpoint gas limit
if stdTx.Msg.Type() == "checkpoint" && stdTx.Msg.Route() == "checkpoint" {
    gasForTx = gasWantedPerCheckpoinTx
}
```

This transaction will store proposed checkpoint on `checkpointBuffer` state instead of actual checkpoint list state.

MsgCheckpointAck

`MsgCheckpointAck` handles successful checkpoint submission. Here `HeaderBlock` is a checkpoint counter;

```
// MsgCheckpointAck represents checkpoint ack transaction if checkpoint is successful
type MsgCheckpointAck struct {
    From        types.HeimdallAddress `json:"from"`
    HeaderBlock uint64                `json:"headerBlock"`
    TxHash      types.HeimdallHash    `json:"tx_hash"`
    LogIndex    uint64                `json:"log_index"`
}
```

For valid `TxHash` and `LogIndex` for the submitted checkpoint, this transaction verifies the following event and validates checkpoint in `checkpointBuffer` state: https://github.com/maticnetwork/contracts/blob/develop/contracts/root/RootChainStorage.sol#L7-L14

```
event NewHeaderBlock(
    address indexed proposer,
    uint256 indexed headerBlockId,
    uint256 indexed reward,
    uint256 start,
    uint256 end,
    bytes32 root
);
```

On successful event verification, it updates the actual count of checkpoint, also known as `ackCount` and clears the `checkpointBuffer`.

MsgCheckpointNoAck

`MsgCheckpointNoAck` handles un-successful checkpoints or offline proposers. This transaction is only valid after `CheckpointBufferTime` has passed from the following events:

- Last successful `ack` transaction
- Last successful `no-ack` transaction

```
// MsgCheckpointNoAck represents checkpoint no-ack transaction
type MsgCheckpointNoAck struct {
    From types.HeimdallAddress `json:"from"`
}
```

This transaction gives the timeout period for the current proposer to send checkpoint/ack before Heimdall chooses a new `proposer` for the next checkpoint.

Parameters

The checkpoint module contains the following parameters:

Key	Type	Default value
CheckpointBufferTime	uint64	1000 * time.Second

CLI Commands

Params

To print all params:

```
heimdallcli query checkpoint params --trust-node
```

Expected Result:

```
checkpoint_buffer_time: 16m40s
```

Send Checkpoint

Following command sends checkpoint transaction on Heimdall:

```
heimdallcli tx checkpoint send-checkpoint \
    --start-block=<start-block> \
    --end-block=<end-block> \
    --root-hash=<root-hash> \
    --account-root-hash=<account-root-hash> \
    --chain-id=<chain-id>
```

Send `ack`

Following command sends ack transaction on Heimdall if checkpoint is successful on Ethereum:

```
heimdallcli tx checkpoint send-ack \
    --tx-hash=<checkpoint-tx-hash>
    --log-index=<checkpoint-event-log-index>
    --header=<checkpoint-index> \
  --chain-id=<chain-id>
```

Send `no-ack`

Following command send no-ack transaction on Heimdall:

```
heimdallcli tx checkpoint send-noack --chain-id <chain-id>
```

REST APIs

Name	Method	Endpoint
It returns the prepared msg for ack checkpoint	POST	/checkpoint/ack
It returns the prepared msg for new checkpoint	POST	/checkpoint/new
It returns the prepared msg for no-ack checkpoint	POST	/checkpoint/no-ack
Checkpoint by number	GET	/checkpoints/<checkpoint-number>
Get current checkpoint buffer state	GET	/checkpoints/buffer
Get checkpoint counts	GET	/checkpoints/count
Get last no-ack details	GET	/checkpoints/last-no-ack
Get latest checkpoint	GET	/checkpoints/latest
All checkpoints	GET	/checkpoints/list
It returns the checkpoint parameters	GET	/checkpoints/parama
It returns the prepared checkpoint	GET	/checkpoints/prepare
Get ack count, buffer, validator set, validator count and last-no-ack details	GET	/overview

More details about the query and response of the above requests can be found here.

All query APIs will provide result in following format:

```
{
    "height": "1",
    "result": {
        ...
    }
}
```---
id: governance
title: Governance
sidebar_label: Governance
description: System with a 1 token - 1 vote basis
keywords:
  - docs
  - matic
  - one token
  - one vote
  - governance
  - heimdall
image: https://matic.network/banners/matic-network-16x9.png

Governance

Heimdall governance works exactly the same as [Cosmos-sdk `x/gov` module](https://docs.cosmos.network/master/modules/gov/).

In this system, holders of the native staking token of the chain can vote on proposals on a `1 token = 1 vote` basis. Here is a list of features the module currently supports:

- **Proposal submission:** Validators can submit proposals with a deposit. Once the minimum deposit is reached, proposal enters voting period. Valdiators that deposited on proposals can recover their deposits once the proposal is rejected or accepted.
- **Vote:** Validators can vote on proposals that reached MinDeposit.

There are deposit period and voting period as params in `gov` module. Minimum deposit has to be achieved before deposit period

```
ends, otherwise proposal will be automatically rejected.

Once minimum deposits reached within deposit period, voting period starts. In voting period, all validators should vote their choices for the proposal. After voting period ends, `gov/Endblocker.go` executes `tally`  function and accepts or rejects proposal based on `tally_params` — `quorum`, `threshold` and `veto`.

Source: [https://github.com/maticnetwork/heimdall/blob/develop/gov/endblocker.go]
(https://github.com/maticnetwork/heimdall/blob/develop/gov/endblocker.go)

There are different types of proposals that can be implemented in Heimdall. As of now, it supports only the **Param change proposal**.

### Param change proposal

Using this type of proposal, validators can change any `params` in any `module` of Heimdall.

Example: change minimum `tx_fees` for the transaction in `auth` module. When the proposal gets accepted, it automatically changes the `params` in Heimdall state. No extra TX is needed.

## CLI Commands

### Query gov params

```go
heimdallcli query gov params --trust-node
```
```
```

This shows all params for governance module.

```
voting_params:
  voting_period: 48h0m0s
tally_params:
  quorum: "334000000000000000"
  threshold: "500000000000000000"
  veto: "334000000000000000"
deposit_parmas:
  min_deposit:
  - denom: matic
    amount:
      i: "10000000000000000000"
  max_deposit_period: 48h0m0s
```

Submit proposal

```
heimdallcli tx gov submit-proposal \
    --validator-id 1 param-change proposal.json \
    --chain-id <heimdall-chain-id>
```

`proposal.json` is a file which includes proposal in json format.

```
{
  "title": "Auth Param Change",
  "description": "Update max tx gas",
  "changes": [
    {
      "subspace": "auth",
      "key": "MaxTxGas",
      "value": "2000000"
    }
  ],
  "deposit": [
    {
      "denom": "matic",
      "amount": "10000000000000000000"
    }
  ]
}
```

Query proposal

To query all proposals:

```
heimdallcli query gov proposals --trust-node
```

To query a particular proposal:
```

```
heimdallcli query gov proposals 1 --trust-node
```

Vote on proposal

To vote on a particular proposal:

```
heimdallcli tx gov vote 1 "Yes" --validator-id 1  --chain-id <heimdal-chain-id>
```

Proposal will be automatically tallied after voting period.

REST APIs

Name	Method	Endpoint
Get all proposals	GET	/gov/proposals
Get proposal details	GET	/gov/proposals/`proposal-id`
Get all votes for the proposal	GET	/gov/proposals/`proposal-id`/votes

---

id: staking title: Staking description: Module that manages validator-related transactions and state keywords: - docs - matic - staking - heimdall - validator image: https://matic.network/banners/matic-network-16x9.png

import useBaseUrl from '@docusaurus/useBaseUrl';

Staking

Staking module manages validator related transactions and state for Heimdall. Note that a validator stakes their tokens on the Ethereum chain and becomes a validator. Respective validators send the transactions on Heimdall using necessary parameters to acknowledge the Ethereum stake change. Once the majority of the validators agree on the change on the stake, this module saves the validator information on Heimdall state.

Key Management

For key management, please refer to Validator key management

Delegation

This module only manages validator staking on Heimdall. The delegation is only available on smart contracts on the Ethereum chain. To optimize the delegation rewards calculation on smart contracts, we are using validator shares (ERC20 per validator).

More details here: Delegation (Validator shares)

Rewards

All rewards are distributed on the Ethereum chain. The validators and delegators claim their rewards or re-stake by simply sending the transaction on `StakeManager.sol`

More details here: Rewards

Messages

<img src={useBaseUrl('img/staking/stake-management-flow.svg')} />

MsgValidatorJoin

`MsgValidatorJoin` handles the staking when a new validator joins the system. Once validator calls `stake` or `stakeFor` in `StakingManager.sol` on Ethereum, and the new `Staked` event is emitted.

Source: https://github.com/maticnetwork/contracts/blob/develop/contracts/staking/StakingInfo.sol#L27-L34

```
/**
 * Staked event - emitted whenever new validator
 *
 * @param signer           Signer address for the validator
 * @param validatorId      Validator id
 * @param activationEpoch  Activation epoch for validator
 * @param amount           Staked amount
 * @param total            Total stake
 * @param signerPubKey     Signer public key (required by Heimdall/Tendermint)
 */
event Staked(
    address indexed signer,
    uint256 indexed validatorId,
```

```
    uint256 indexed activationEpoch,
    uint256 amount,
    uint256 total,
    bytes signerPubkey
);
```

`activationEpoch` is the checkpoint count from where a validator will become active on Heimdall.

Stake call on smart contract fails if slots are unavailable. Validator slots are the way to restrict a number of validators in the system. Slots are managed on Ethereum smart contracts.

Here is `ValidatorJoin` message for Heimdall transaction:

```
type MsgValidatorJoin struct {
    From         hmTypes.HeimdallAddress `json:"from"`
    ID           hmTypes.ValidatorID     `json:"id"`
    SignerPubKey hmTypes.PubKey          `json:"pub_key"`
    TxHash       hmTypes.HeimdallHash    `json:"tx_hash"`
    LogIndex     uint64                  `json:"log_index"`
}
```

MsgStakeUpdate

`MsgStakeUpdate` handles the stake update when a validator the re-stakes or new delegation comes in. In either case, the new `StakeUpdate` event is emitted.

```
/**
 * Stake update event - emitted whenever stake gets updated
 *
 * @param validatorId      Validator id
 * @param newAmount        New staked amount
 */
event StakeUpdate(
    uint256 indexed validatorId,
    uint256 indexed newAmount
);
```

Here is `MsgStakeUpdate` message for Heimdall transaction:

```
// MsgStakeUpdate represents stake update
type MsgStakeUpdate struct {
    From     hmTypes.HeimdallAddress `json:"from"`
    ID       hmTypes.ValidatorID     `json:"id"`
    TxHash   hmTypes.HeimdallHash    `json:"tx_hash"`
    LogIndex uint64                  `json:"log_index"`
}
```

MsgValidatorExit

`MsgValidatorExit` handles the validator exit process after a validator initiates the exit process on Ethereum. It emits `SignerUpdate` event.

```
/**
 * Unstake init event - emitted whenever validator initiates the exit
 *
 * @param user                Signer
 * @param validatorId         Validator id
 * @param deactivationEpoch   Deactivation epoch for validator
 * @param amount              Unstaked amount
 */
event UnstakeInit(
    address indexed user,
    uint256 indexed validatorId,
    uint256 deactivationEpoch,
    uint256 indexed amount
);
```

Here is `MsgValidatorExit` message for Heimdall transaction:

```
type MsgValidatorExit struct {
    From     hmTypes.HeimdallAddress `json:"from"`
    ID       hmTypes.ValidatorID     `json:"id"`
    TxHash   hmTypes.HeimdallHash    `json:"tx_hash"`
    LogIndex uint64                  `json:"log_index"`
}
```

MsgSignerUpdate

`MsgSignerUpdate` handles the signer update when a validator updates signer key on Ethereum. It emits `SignerUpdate` event.

```
/**
 * Signer change event - emitted whenever signer key changes
 *
 * @param validatorId      Validator id
 * @param oldSigner        Current old signer
 * @param newSigner        New signer
 * @param signerPubkey     New signer public key
 */
event SignerChange(
    uint256 indexed validatorId,
    address indexed oldSigner,
    address indexed newSigner,
    bytes signerPubkey
);
```

Here is `MsgSignerUpdate` message for Heimdall transaction:

```
// MsgSignerUpdate signer update struct
type MsgSignerUpdate struct {
    From            hmTypes.HeimdallAddress `json:"from"`
    ID              hmTypes.ValidatorID     `json:"id"`
    NewSignerPubKey hmTypes.PubKey          `json:"pubKey"`
    TxHash          hmTypes.HeimdallHash    `json:"tx_hash"`
    LogIndex        uint64                  `json:"log_index"`
}
```

CLI Commands

Validator details

By signer address

```
heimdallcli query staking validator-info \
    --validator=<signer-address> \
    --chain-id <chain-id>
```

This command should display the following output:

```
{
    "ID":1,
    "startEpoch":0,
    "endEpoch":0,
    "power":10,

"pubKey":"0x04b12d8b2f6e3d45a7ace12c4b2158f79b95e4c28ebe5ad54c439be9431d7fc9dc1164210bf6a5c3b8523528b931e772c86a307e8cff4b725e6b4a7
    "signer":"0x6c468cf8c9879006e22ec4029696e005c2319c9d",
    "last_updated":0,
    "accum":0
}
```

By validator address

```
heimdallcli query staking validator-info \
    --id=<validator-id> \
    --chain-id=<chain-id>
```

This command should display the following output:

```
{
    "ID":1,
    "startEpoch":0,
    "endEpoch":0,
    "power":10,

"pubKey":"0x04b12d8b2f6e3d45a7ace12c4b2158f79b95e4c28ebe5ad54c439be9431d7fc9dc1164210bf6a5c3b8523528b931e772c86a307e8cff4b725e6b4a7
    "signer":"0x6c468cf8c9879006e22ec4029696e005c2319c9d",
    "last_updated":0,
    "accum":0
}
```

Validator join

This command sends validator join command through CLI:

```
heimdallcli tx staking validator-join \
    --signer-pubkey <signer-public-key> \
    --tx-hash <tx-hash>   \
    --log-index <log-index> \
    --chain-id <chain-id>
```

`tx-hash` value must be the same as Ethereum TX hash which emitted `Staked` event and `log-index` must be the same at which index the event is emitted.

REST APIs

Name	Method	Endpoint
Get Heimdall validator set	GET	/staking/validator-set
Get validator details	GET	/staking/validator/validator-id

All query APIs will result in following format:

```
{
    "height": "1",
    "result": {
        ...
    }
}
```---
id: overview
title: Overview
description: Heimdall is the heart of the Polygon network
keywords:
  - docs
  - matic
  - polygon
  - heimdall
  - cosmos
  - peppermint
image: https://matic.network/banners/matic-network-16x9.png

Heimdall

Heimdall is the heart of the Polygon network. It manages validators, block producer selection, spans, the state-sync mechanism between Ethereum and Matic and other essentials aspects of the system.

It uses the **Cosmos-SDK** and a forked version of Tendermint, called **Peppermint**. Here is the Peppermint source: https://github.com/maticnetwork/tendermint/tree/peppermint

Heimdall removes some of the modules from Cosmos-SDK, but mostly uses a customised version of it while following the same pattern.---
id: peppermint
title: Peppermint
description: Peppermint is a modified Ethereum-compatible Tendermint
keywords:
  - docs
  - matic
  - polygon
  - tendermint
  - peppermint
image: https://matic.network/banners/matic-network-16x9.png

Peppermint is a modified Tendermint. It is changed to make it compatible with Ethereum addresses and verifiable on Ethereum chain.

Overview

1. Changes to signature scheme
2. Changes to `vote` to make it verifiable on Ethereum smart contract
3. Changes to `vote` encoding scheme

Peppermint uses `secp256k1` signature scheme to verify Tendermint votes on solidity smart contract.

Source: https://github.com/maticnetwork/tendermint/blob/peppermint/crypto/secp256k1/secp256k1_nocgo.go

```
It adds `Data` field into `Vote` and `Proposal` struct to get `hash` for transactions in the block. On smart contract, it checks
if `Data` matches with checkpoint data hash and majority (⅔+1) of validator signatures. The idea is to verify if majority of the
validator set agrees on transaction in the contract.

Peppermint uses RLP to get `Vote` bytes instead of Amino encoding. Here `Data` is `Txs.Hash()` for the block.

Source: [https://github.com/maticnetwork/tendermint/blob/peppermint/types/canonical.go]
(https://github.com/maticnetwork/tendermint/blob/peppermint/types/canonical.go)

```go
// [peppermint] create RLP vote to decode in contract
type CanonicalRLPVote struct {
    ChainID string
    Type    byte
    Height  uint
    Round   uint
    Data    []byte
}
```
```

And using RLP encoding lib to get byte data for signature on Vote.

Source: https://github.com/maticnetwork/tendermint/blob/peppermint/types/vote.go#L75-L82

```go
func (vote *Vote) SignBytes(chainID string) []byte {
    // [peppermint] converted from amino to rlp
    bz, err := rlp.EncodeToBytes(CanonicalizeVote(chainID, vote))
    if err != nil {
        panic(err)
    }
    return bz
}
```

Complete Source: https://github.com/maticnetwork/tendermint

id: stdtx title: StdTx description: A standard way to wrap a Msg with Fee and Signatures keywords: - docs - matic - polygon - StdTx - wrap a msg with fee image: https://matic.network/banners/matic-network-16x9.png

StdTx

Heimdall's `StdTx` is not using `Fee` for each transaction. We have very limited types of the transactions and as end-users won't be deploying any kind of contracts on Heimdall, it is using fixed fee model for the transaction.

Check here: https://github.com/maticnetwork/heimdall/blob/master/auth/ante.go#L32

```go
// StdTx is a standard way to wrap a Msg with Fee and Signatures.
type StdTx struct {
  Msg       sdk.Msg      `json:"msg" yaml:"msg"`
  Signature StdSignature `json:"signature" yaml:"signature"`
  Memo      string       `json:"memo" yaml:"memo"`
}
```

Source: https://github.com/maticnetwork/heimdall/blob/master/auth/types/stdtx.go--- id: transactions title: Transactions description: What are transactions and when they are used keywords: - docs - matic - polygon - Transactions image: https://matic.network/banners/matic-network-16x9.png

Transactions

Transactions are comprised of metadata held in contexts and messages that trigger state changes within a module, through the module's Handler.

When users want to interact with an application and make state changes (e.g. sending coins), they create transactions. Each of a transaction's `message` must be signed using the private key associated with the appropriate account before the transaction is broadcasted to the network. A transaction must then be included in a block, validated, and then approved by the network through the consensus process. To read more about the lifecycle of a transaction, click here.

Type Definition

Transaction objects are SDK types that implement the `Tx` interface.

```go
type Tx interface {
    // Gets all the transaction's messages.
```

```
    GetMsgs() []Msg

    // ValidateBasic does a simple and lightweight validation check that doesn't
    // require access to any other information.
    ValidateBasic() Error
}
```

More details on Transactions: https://docs.cosmos.network/main/core/transactions.html

id: types title: Types description: Description of HeimdallAddress, Pubkey, & HeimdallHash keywords: - docs - matic - HeimdallAddress - polygon - Pubkey - HeimdallHash image: https://matic.network/banners/matic-network-16x9.png

Types

HeimdallAddress

`HeimdallAddress` represents address on Heimdall. It uses Ethereum's common library for Address. Length of this address is 20 bytes.

```
// HeimdallAddress represents Heimdall address
type HeimdallAddress common.Address
```

PubKey

It represents public key used in Heimdall, `ecdsa` compatible uncompressed public key.

```
// PubKey pubkey
type PubKey [65]byte
```

HeimdallHash

It represents hash in Heimdall. It uses Ethereum's hash for the same.

```
// HeimdallHash represents heimdall address
type HeimdallHash common.Hash
```---
id: validator-key-management
title: Validator Key Management
description: Signer and Owner Key validator management
keywords:
  - docs
  - matic
  - polygon
  - Validator Key Management
  - signer
  - owner
image: https://matic.network/banners/matic-network-16x9.png

Validator Key Management

Each validator uses two keys to manage validator related activities on Polygon. The Signer key is kept on the node and is generally considered a `hot` wallet, whereas the Owner key is supposed to kept very secure, is used infrequently, and is generally considered a `cold` wallet. The staked funds are controlled by the Owner key.

This separation of responsibilities has been done to ensure an efficient tradeoff between security and ease of use. Both keys are Ethereum compatible addresses and work exactly the same manner. And yes, it is possible to have same Owner and Signer keys.

Signer Key

The signer key is an address that is used for signing Heimdall blocks, checkpoints, and other signing related activities. This key's private key will be on the Validator node for signing purposes. It cannot manage stake, rewards or delegations.

The validator must keep two types of balances on this address:

- Matic tokens on Heimdall (through Topup transactions) to perform validator responsibilities on Heimdall
- ETH on Ethereum chain to send checkpoints on Ethereum

Owner Key

The owner key is an address that is used for staking, re-stake, changing the signer key, withdraw rewards and manage delegation related parameters on the Ethereum chain. The private key for this key must be secure at all cost.

```
All transactions through this key will be performed on the Ethereum chain.

## Signer Change

Following event is generated in case of signer change on Ethereum chain on `StakingInfo.sol`:
[https://github.com/maticnetwork/contracts/blob/develop/contracts/staking/StakingInfo.sol]
(https://github.com/maticnetwork/contracts/blob/develop/contracts/staking/StakingInfo.sol)

```go
// Signer change
event SignerChange(
  uint256 indexed validatorId,
  address indexed oldSigner,
  address indexed newSigner,
  bytes signerPubkey
);
```
```

Heimdall bridge processes these events and sends transactions on Heimdall to change state based on the events.--- id: validators title: Heimdall Validators description: Heimdall's structure for validators keywords: - docs - matic - polygon - heimdall - validators image: https://matic.network/banners/matic-network-16x9.png

Heimdall Validators

Validators are an essential part of the Heimdall. Heimdall has an option to change validators at the end of each block. It is called `EndBlocker` which is a part of Cosmos-SDK: https://docs.cosmos.network/master/building-modules/beginblock-endblock.html

Heimdall uses the following structure for Validators:

```go
// Validator for Heimdall
type Validator struct {
    ID          ValidatorID     `json:"ID"`
    StartEpoch  uint64          `json:"startEpoch"`
    EndEpoch    uint64          `json:"endEpoch"`
    VotingPower int64           `json:"power"`
    PubKey      PubKey          `json:"pubKey"`
    Signer      HeimdallAddress `json:"signer"`
    LastUpdated string          `json:"last_updated"`

    ProposerPriority int64 `json:"accum"`
}

// ValidatorID  validator ID and helper functions
type ValidatorID uint64
```

Here `StartEpoch` and `EndEpoch` are checkpoint numbers in between validator will be active as a validator. In `EndBlocker` for the app, Heimdall gets all active validators and updates the current validator set in the state.

In the end, it returns validator updates for the Tendermint.

Source: https://github.com/maticnetwork/heimdall/blob/develop/app/app.go#L500-L542

```go
// --- Start update to new validators
currentValidatorSet := app.StakingKeeper.GetValidatorSet(ctx)
allValidators := app.StakingKeeper.GetAllValidators(ctx)
ackCount := app.CheckpointKeeper.GetACKCount(ctx)

// get validator updates
setUpdates := helper.GetUpdatedValidators(
    &currentValidatorSet, // pointer to current validator set -- UpdateValidators will modify it
    allValidators,        // All validators
    ackCount,             // ack count
)

if len(setUpdates) > 0 {
    // create new validator set
    if err := currentValidatorSet.UpdateWithChangeSet(setUpdates); err != nil {
        // return with nothing
        logger.Error("Unable to update current validator set", "Error", err)
        return abci.ResponseEndBlock{}
    }

    // increment proposer priority
    currentValidatorSet.IncrementProposerPriority(1)
```

```
    // save set in store
    if err := app.StakingKeeper.UpdateValidatorSetInStore(ctx, currentValidatorSet); err != nil {
        // return with nothing
        logger.Error("Unable to update current validator set in state", "Error", err)
        return abci.ResponseEndBlock{}
    }

    // convert updates from map to array
    for _, v := range setUpdates {
        tmValUpdates = append(tmValUpdates, abci.ValidatorUpdate{
            Power:  int64(v.VotingPower),
            PubKey: v.PubKey.ABCIPubKey(),
        })
    }
}

// send validator updates to peppermint
return abci.ResponseEndBlock{
    ValidatorUpdates: tmValUpdates,
}
```---
id: eip1559
title: How to Send EIP1559 Transactions
sidebar_label: EIP1559 Transactions
description: "Learn about EIP1559 transactions."
keywords:
  - docs
  - polygon
  - matic
  - transaction
  - eip1559
  - ethereum improvement proposal
image: https://wiki.polygon.technology/img/polygon-logo.png
slug: eip1559-transactions/how-to-send-eip1559-transactions

The [London hard fork](https://blog.polygon.technology/eip-1559-upgrades-are-going-live-on-polygon-mainnet/) introduced a new [EIP](https://eips.ethereum.org/EIPS/eip-1559) that modifies how gas estimation and costs work for transactions on Polygon.

Due to this, there is a change in how the transaction object is formed when sending transactions on Polygon. A new transaction type called **Type 2 Transaction** has been introduced. The legacy type transactions will still be compatible but it is recommended to shift to the new style. You can navigate to the end of this document to directly peek into the code.

How legacy transactions (Type 0) work

When you submit a transaction, you also send a `gasPrice` which is an amount you are offering to pay per gas consumed. Then, when you submit the transaction, miners can decide to include your transaction or not based on your `gasPrice` bid. Miners will prioritize the highest gas prices.

Sending Type 2 transactions with EIP1559

It is a similar concept, the `gasPrice` will be split into a `baseFee` and a `priorityFee`.

Every transaction needs to pay the `base fee`, which is calculated based on how full the previous block was. Transactions can also offer the miner a `priorityFee` to incentivize the miner to include the transaction in the block.

Sending Legacy transactions

Only the `gasPrice` needed to be mentioned in the legacy transaction prior to the London fork.

The following code example shows sending transaction using a type 0 transaction:

```jsx
const sendLegacyTransaction = async () => {
    const web3 = new Web3('https://polygon-rpc.com');

    await web3.eth.sendTransactions({
        from: 0x05158d7a59FA8AC5007B3C8BabAa216568Fd32B3,
        to: 0xD7Fbe63Db5201f71482Fa47ecC4Be5e5B125eF07,
        value: 1000000000000000000,
        gasPrice: 200000000000
    })
}
```

Sending EIP1559 transactions

Add maxPriorityFeePerGas field

The closest analogy to the `gas` : `gasPrice` combination is `gas` : `maxPriorityFeePerGas` . Since the `baseFee` needs to be paid regardless, we can just submit a bid on the *"tip"* for the miner. Note that the Polygon Gas Station V2 can be used to get the gas fee estimates.

The following code example shows sending transaction in Type 2 method:

```
// Example for
const sendEIP1559Transaction = async () => {
    const web3 = new Web3('https://polygon-rpc.com');

    await web3.eth.sendTransactions({
        from: 0xFd71Dc9721d9ddCF0480A582927c3dCd42f3064C,
        to: 0x8C400f640447A5Fc61BFf7FdcE00eCf20b85CcAd,
        value: 100000000000000000,
        maxPriorityFeePerGas: 40000000000
    })
}
```

The Polygon Gas Station V2 can be used to get the gas fee estimates.

Polygon Gas Station V2 Endpoint:

```
https://gasstation.polygon.technology/v2
```

Polygon Gas Station V2 Response:

```
{
    "safeLow": {
        "maxPriorityFee": 37.181444553750005,
        "maxFee": 326.2556979087
    },
    "standard": {
        "maxPriorityFee": 49.575259405,
        "maxFee": 435.00759721159994
    },
    "fast": {
        "maxPriorityFee": 61.96907425625,
        "maxFee": 543.7594965144999
    },
    "estimatedBaseFee": 275.308812719,
    "blockTime": 6,
    "blockNumber": 23948420
}
```

See also

Please read the following articles to get a better understanding of sending EIP-1559 transactions:

- How to Send Transactions with EIP 1559, this tutorial will walk you through both the legacy and new (EIP-1559) way to estimate gas and send transactions.
- Learn how to send an EIP-1559 transaction using ethers.js

---

id: account-abstraction title: Account Abstraction sidebar_label: Overview description: "Learn about how account abstraction can improve transaction experience" keywords: - docs - polygon - matic - transaction - account abstraction - meta transaction - eip4337 - EIP-4337 - ERC-4337 image: https://wiki.polygon.technology/img/polygon-logo.png slug: account-abstraction

Account Abstraction is a blockchain technology that enables users to utilize smart contracts as their accounts. While the default account for most users is an Externally Owned Account (EOA), which is controlled by an external private key, it requires users to have a considerable understanding of blockchain technology to use them securely. Fortunately, smart contract accounts can create superior user experiences.

Benefits

Contract accounts offer numerous benefits, including:

- **Arbitrary verification logic:** Rather than use an external private key, contract accounts can use any arbitrary signature type. This feature supports single and multi-signature verification and any signature scheme.
- **Sponsored transactions:** Users can pay transaction fees in ERC-20 tokens or create their fee logic, including sponsoring transaction fees on their app.
- **Account security:** Contract accounts enable social recovery and security features such as time-locks and withdraw limits.
- **Atomic multi-operations:** Users can perform multiple operations simultaneously, such as trading in a single click instead of approving and swapping separately.

Using Account Abstraction on Polygon

There are two primary ways users can use account abstraction on Polygon: by sending ERC-4337 transactions or with third party meta transaction services.

ERC-4337

ERC-4337, also known is EIP-4337, brings account abstraction to the Polygon ecosystem and all EVM-compatible chains.

Meta Transactions

Meta transactions are bespoke third party services for achieving account abstraction.

id: eip4337 title: ERC-4337 sidebar_label: ERC-4337 Transactions description: "Learn about how to use EIP-4337 on the Polygon network" keywords: - docs - polygon - matic - transaction - account abstraction - eip4337 - EIP-4337 - ERC-4337 image: https://wiki.polygon.technology/img/polygon-logo.png slug: eip-4337

The ERC-4337 standard, also known as EIP-4337, allows developers to achieve account abstraction on the Polygon Blockchain. This page provides a simplified overview of the different components of ERC-4337 and how they work together.

The ERC-4337 standard consists of four main components: `UserOperation`, `Bundler`, `EntryPoint`, and `Contract Account`. Optional components include `Paymasters` and `Aggregators`.

Building ERC-4337 Transactions

ERC-4337 transactions are called `UserOperations` to avoid confusion with the regular transaction type. `UserOperations` are pseudo-transaction objects that are used to execute transactions with contract accounts. ERC-4337 transactions have to be sent to nodes that include ERC-4337 bundlers.

The `UserOperation` object has a few fields.

Field	Type	Description
`sender`	`address`	The address of the smart contract account
`nonce`	`uint256`	Anti-replay protection
`initCode`	`bytes`	Code used to deploy the account if not yet on-chain
`callData`	`bytes`	Data that's passed to the `sender` for execution
`callGasLimit`	`uint256`	Gas limit for execution phase
`verificationGasLimit`	`uint256`	Gas limit for verification phase
`preVerificationGas`	`uint256`	Gas to compensate the bundler
`maxFeePerGas`	`uint256`	Similar to EIP-1559 max fee
`maxPriorityFeePerGas`	`uint256`	Similar to EIP-1559 priority fee
`paymasterAndData`	`bytes`	`Paymaster Contract` address and any extra data required for verification and execution
`signature`	`bytes`	Used to validate a `UserOperation` along with the `nonce` during verification

ERC-4337 Wallets

Users must have an ERC-4337 smart contract account to validate `UserOperations`. The core interface for an ERC-4337 wallet is:

```
interface IAccount {
  function validateUserOp
      (UserOperation calldata userOp, bytes32 userOpHash, address aggregator, uint256 missingAccountFunds)
      external returns (uint256 sigTimeRange);
}
```

Resources

- ERC-4337 proposal is the link to the official proposal and technical specification.
- @account-abstraction SDK is an npm package for using ERC-4337 developed by the authors of the proposal.
- Stackup provides node services with ERC-4337 bundlers and other ERC-4337 infrastructure.

---

id: meta-transactions title: Meta Transactions sidebar_label: Meta Transactions description: Learn about meta transactions and how you can use them. keywords: - docs - polygon - matic - transactions - meta transactions - gasless image: https://matic.network/banners/matic-network-16x9.png slug: meta-transactions

Daily smart contract calls are at their highest, hitting around 2.5 to 3 million per day. DApps are starting to realize their utility but are becoming victims of their success or others' success due to gas fees. Not to mention, the onboarding hurdles of users and the challenges of current UX are no easy fix.

Servicing Smart Contracts

By design, smart contracts are deterministic state machines that execute when transaction fees are paid to service the contract's logic by using the network's computational resources. This is accomplished by a gas-metered model on Ethereum (and Polygon).

The Current State of Transacting

There are limitations to this traditional transaction model on Ethereum (and other blockchains alike). A common limitation is a user not having the means to pay for gas. By default, the sender of the transaction acts as the payer, as these behaviors are coupled, so if a user attempts to create and send a transaction, they are responsible for the associated gas fees. Likewise, if a user builds, interacts with, or runs a dApp, the user is required to pay gas.

It is unrealistic to expect the average user to buy crypto and pay for gas to interact with an application. What can be done to address this is to decouple the sender of a transaction from acting as a payer, enabling the opportunity to scale transaction execution and initiate a seamless transacting experience.

Instead of direct transaction execution, a middleware would exist (via a third party) to handle the gas. This is where meta transactions come in.

What are Meta Transactions?

Meta transactions allow anyone to interact with the blockchain. They do not require users to have tokens to pay for the network's services through transaction fees. This is done by decoupling the sender of a transaction and the payer of gas.

A solution that can onboard new users and helps current ones.

The executor of a transaction acts as a sender. Rather than spending gas, they only create a transaction request by signing their intended action (the transaction parameters) with their private key. The meta transaction is a regular Ethereum transaction that includes additional parameters to craft the meta transaction.

The signed transaction parameters are passed onto a secondary network, which acts as a relayer. While there are different schemes for this, relayers would generally choose which transactions are worth submitting by validating the transaction (e.g., being relevant to the dApp). Upon validation, the relayer will wrap the request (the signed message) into an actual transaction (which means paying the gas fee) and broadcast it to the network, where the contract unwraps the transaction by validating the original signature and executes it on behalf of the user.

:::note The words meta and batch may be analogous to some

To clarify: a meta transaction is different from a batch transaction, where a batch transaction is a transaction that can send multiple transactions at once and are then executed from a single sender (single nonce specified) in sequence.

:::

In summary, meta transactions are a design pattern where:

- A user (sender) signs a request with their private key and sends it to a relayer
- The relayer wraps the request into a tx and sends it to a contract
- The contract unwraps the tx and executes it

Native transactions imply that the "sender" is also the "payer". When taking the "payer" away from the "sender", the "sender" becomes more like an "intender" - the sender shows the intent of the transaction they would like executed on the blockchain by signing a message containing specific parameters related to their message, and not an entirely constructed transaction.

Use Cases

One can imagine the capabilities of meta transactions for scaling dApps and interactions with smart contracts. Not only can a user create a gasless transaction, but they can also do so many times, and with an automation tool, meta transactions can influence the next wave of applications for practical use cases. Meta transactions enable real utility in smart contract logic, which is often limited because of gas fees and the interactions required on-chain.

Example with voting

A user wants to participate in on-chain governance, and they intend to vote for a particular outcome via a voting contract. The user would sign a message which states the user's decision in a vote in this particular contract. Traditionally, they would need to pay a gas fee for interacting with the contract (and know how to interact with the contract), but instead, they can sign a meta transaction (off-chain) with the necessary information for their vote and pass it to a relayer which would execute the transaction on their behalf.

The signed message gets sent to a relayer (the signed tx params about the voting information). The relayer validates that this transaction is a priority vote, wraps the voting request into an actual transaction, pays the gas fees, and broadcasts it to the voting contract. Everything checks out on the voting contract's end, and the vote executes on behalf of the user.

Try Them Out

Assuming your familiarity with the different approaches you can take to integrate meta transactions in your dApp, and depending on whether you're migrating to meta transactions or building fresh dApp on using it.

To integrate your dApp with Meta Transactions on Polygon, you can choose to go with one of the following relayers or spin up a custom solution:

- Biconomy
- Gas Station Network (GSN)
- Infura
- Gelato

---

id: network-agnostics title: Network Agnostic Transactions sidebar_label: Network Agnostic Transactions description: "Integrate Network Agnostic Transactions in your dApp." keywords: - docs - matic image:

https://matic.network/banners/matic-network-16x9.png

Goal

Execute transactions on Polygon chain, without changing provider on MetaMask (this tutorial caters to metamask's inpage provider, can be modified to execute transactions from any other provider)

Under the hood, user signs on an intent to execute a transaction, which is relayed by a simple relayer to execute it on a contract deployed on Polygon chain.

What is enabling transaction execution?

The client that the user interacts with (web browser, mobile apps, etc) never interacts with the blockchain, instead it interacts with a simple relayer server (or a network of relayers), similar to the way GSN or any meta-transaction solution works (see: Meta Transactions: An Introduction).

For any action that requires blockchain interaction,

- Client requests an EIP712 formatted signature from the user
- The signature is sent to a simple relayer server (should have a simple auth/spam protection if used for production, or biconomy's mexa sdk can be used: https://github.com/bcnmy/mexa-sdk)
- The relayer interacts with the blockchain to submit user's signature to the contract. A function on the contract called `executeMetaTransaction` processes the signature and executes the requested transaction (via an internal call).
- The relayer pays for the gas making the transaction effectively free 🤑

Integrate Network Agnostic Transactions in your dApp

- Choose between a custom simple relayer node/biconomy.
  - For biconomy, setup a dapp from the dashboard and save the api-id and api-key, see: Tutorial: Biconomy or https://docs.biconomy.io/

    Steps:

    1. Let's Register our contracts to biconomy dashboard
       1. Visit the official documents of biconomy.
       2. While registering the dapp, select `Polygon Mumbai`
    2. Copy the `API key` to use in frontend
    3. And Add function `executeMetaTransaction` in Manage-Api and make sure to enable meta-tx. (Check 'native-metatx' option)
  - If you'd like to use your own custom API that sends signed transactions on the blockchain, you can refer to the server code here: https://github.com/angelagilhotra/ETHOnline-Workshop/tree/master/2-network-agnostic-transfer
- Make sure that the contract you'd like to interact with inherits from `NativeMetaTransactions` - 👀 peep into `executeMetaTransaction` function in the contract.
- Link: https://github.com/maticnetwork/pos-portal/blob/34be03cfd227c25b49c5791ffba6a4ffc9b76036/flat/ChildERC20.sol#L1338

```
let data = await web3.eth.abi.encodeFunctionCall({
    name: 'getNonce',
    type: 'function',
    inputs: [{
        name: "user",
        type: "address"
      }]
  }, [accounts[0]]);

  let _nonce = await web3.eth.call ({
    to: token["80001"],
    data
  });

  const dataToSign = getTypedData({
    name: token["name"],
    version: '1',
    salt: '0x0000000000000000000000000000000000000000000000000000000000013881',
    verifyingContract: token["80001"],
    nonce: parseInt(_nonce),
    from: accounts[0],
    functionSignature: functionSig
  });

  const msgParams = [accounts[0], JSON.stringify(dataToSign)];

  let sig = await eth.request ({
    method: 'eth_signTypedData_v3',
    params: msgParams
  });
```

- Once you have a relayer and the contracts setup, what is required is for the client to be able to fetch an EIP712 formatted signature and simply call the API with the required parameters

  ref: https://github.com/angelagilhotra/ETHOnline-Workshop/blob/6b615b8a4ef00553c17729c721572529303c8e1b/2-network-agnostic-transfer/sign.js#L47

```
let data = await web3.eth.abi.encodeFunctionCall({
    name: 'getNonce',
    type: 'function',
    inputs: [{
        name: "user",
        type: "address"
      }]
  }, [accounts[0]]);

  let _nonce = await web3.eth.call ({
    to: token["80001"],
    data
  });

  const dataToSign = getTypedData({
    name: token["name"],
    version: '1',
    salt: '0x0000000000000000000000000000000000000000000000000000000000000013881',
    verifyingContract: token["80001"],
    nonce: parseInt(_nonce),
    from: accounts[0],
    functionSignature: functionSig
  });
  const msgParams = [accounts[0], JSON.stringify(dataToSign)];

  let sig = await eth.request ({
    method: 'eth_signTypedData_v3',
    params: msgParams
  });
```

Calling the API, ref: https://github.com/angelagilhotra/ETHOnline-Workshop/blob/6b615b8a4ef00553c17729c721572529303c8e1b/2-network-agnostic-transfer/sign.js#L110

```
const response = await request.post(
    'http://localhost:3000/exec', {
      json: txObj,
    },
    (error, res, body) => {
      if (error) {
        console.error(error)
        return
      }
      document.getElementById(el).innerHTML =
      `response:`+ JSON.stringify(body)
    }
  )
```

If using Biconomy, the following should be called:

```
const response = await request.post(
    'https://api.biconomy.io/api/v2/meta-tx/native', {
      json: txObj,
    },
    (error, res, body) => {
      if (error) {
        console.error(error)
        return
      }
      document.getElementById(el).innerHTML =
      `response:`+ JSON.stringify(body)
    }
  )
```

where the `txObj` should look something like:

```
{
    "to": "0x2395d740789d8C27C139C62d1aF786c77c9a1Ef1",
    "apiId": <API ID COPIED FROM THE API PAGE>,
    "params": [
```

```
        "0x2173fdd5427c99357ba0dd5e34c964b08079a695",
        "0x2e1a7d4d000000000000000000000000000000000000000000000000000000000000000a",
        "0x42da8b5ac3f1c5c35c3eb38d639a780ec973744f11ff75b81bbf916300411602",
        "0x32bf1451a3e999b57822bc1a9b8bfdfeb0da59aa330c247e4befafa997a11de9",
        "27"
    ],
    "from": "0x2173fdd5427c99357ba0dd5e34c964b08079a695"
}
```

- If you use the custom API it executes the `executeMetaTransaction` function on the contract:

  (ref: https://github.com/angelagilhotra/ETHOnline-Workshop/blob/6b615b8a4ef00553c17729c721572529303c8e1b/2-network-agnostic-transfer/server/index.js#L40)

```
try {
    let tx = await contract.methods.executeMetaTransaction(
      txDetails.from, txDetails.fnSig, r, s, v
    ).send ({
      from: user,
      gas: 800000
    })
    req.txHash = tx.transactionHash
  } catch (err) {
    console.log (err)
    next(err)
  }
```

  is using biconomy, the client side call looks like:

```
// client/src/App.js
import React from "react";
import Biconomy from "@biconomy/mexa";

const getWeb3 = new Web3(biconomy);
biconomy
    .onEvent(biconomy.READY, () => {
      // Initialize your dapp here like getting user accounts etc
      console.log("Mexa is Ready");
    })
    .onEvent(biconomy.ERROR, (error, message) => {
      // Handle error while initializing mexa
            console.error(error);
    });

/**
* use the getWeb3 object to define a contract and calling the function directly
*/
```

---

id: architecture title: Architecture description: Ethereum, Heimdall and Bor layers keywords: - docs - matic - polygon - architecture - validator slug: architecture image: https://wiki.polygon.technology/img/polygon-logo.png

import useBaseUrl from '@docusaurus/useBaseUrl';

The Polygon Network is broadly divided into three layers:

- **Ethereum layer** — a set of contracts on the Ethereum mainnet.
- **Heimdall layer** — a set of proof-of-stake Heimdall nodes running in parallel to the Ethereum mainnet, monitoring the set of staking contracts deployed on the Ethereum mainnet, and committing the Polygon Network checkpoints to the Ethereum mainnet. Heimdall is based on Tendermint.
- **Bor layer** — a set of block-producing Bor nodes shuffled by Heimdall nodes. Bor is based on Go Ethereum.

<img src={useBaseUrl("img/staking/architecture.png")} />

Staking smart contracts on Ethereum

To enable the Proof of Stake (PoS) mechanism on Polygon, the system employs a set of staking management contracts on the Ethereum mainnet.

The staking contracts implement the following features:

- The ability for anyone to stake MATIC tokens on the staking contracts on the Ethereum mainnet and join the system as a validator.
- Earn staking rewards for validating state transitions on the Polygon Network.
- Save checkpoints on the Ethereum mainnet.

The PoS mechanism also acts as a mitigation to the data unavailability problem for the Polygon sidechains.

Heimdall (validation layer)

Heimdall layer handles the aggregation of blocks produced by Bor into a Merkle tree and publishing the Merkle root periodically to the root chain. The periodic publishing of snapshots of Bor are called checkpoints.

For every few blocks on Bor, a validator on the Heimdall layer:

1. Validates all the blocks since the last checkpoint.
2. Creates a Merkle tree of the block hashes.
3. Publishes the Merkle root hash to the Ethereum mainnet.

Checkpoints are important for two reasons:

1. Providing finality on the root chain.
2. Providing proof of burn in withdrawal of assets.

An overview of the process:

- A subset of active validators from the pool is selected to act as block producers for a span. These block producers are responsible for creating blocks and broadcasting the created blocks on the network.
- A checkpoint includes the Merkle root hash of all blocks created during any given interval. All nodes validate the Merkle root hash and attach their signature to it.
- A selected proposer from the validator set is responsible for collecting all signatures for a particular checkpoint and committing the checkpoint on the Ethereum mainnet.
- The responsibility of creating blocks and proposing checkpoints is variably dependent on a validator's stake ratio in the overall pool.

See also Heimdall architecture.

Bor (block producer layer)

Bor is Polygon's sidechain block producer — the entity responsible for aggregating transactions into blocks.

Bor block producers are a subset of the validators and are shuffled periodically by the Heimdall validators.

See also Bor architecture.

id: bridge title: Bridge description: Work In Progress keywords: - docs - polygon - matic - bridge slug: bridge image: https://wiki.polygon.technology/img/polygon-logo.png

Bridge

:::info

This section is a work in progress.

:::--- id: bor-chain title: Bor Chain description: Polygon network block producer keywords: - docs - polygon - matic - bor - bor chain - go ethereum - block producer slug: bor-chain image: https://wiki.polygon.technology/img/polygon-logo.png

The Bor node or the block producer implementation is the EVM-compatible blockchain operator.

Currently, Bor is a basic Geth implementation with custom changes done to the consensus algorithm.

Block producers are chosen from the validator set and are shuffled using historical Ethereum block hashes.

See also Bor architecture.

id: checkpoint-mechanism title: Checkpoint Mechanism sidebar_label: Checkpoints description: Checkpointing the system state to the Ethereum mainnet keywords: - docs - matic - polygon - checkpoint - ethereum - mainnet slug: checkpoint-mechanism image: https://wiki.polygon.technology/img/polygon-logo.png

:::info Polygon is not a Layer 1 platform

Polygon depends on the Ethereum Mainnet as its Layer 1 Settlement Layer. All staking mechanics need to be in sync with the contracts on the Ethereum mainnet.

:::

Proposers for a checkpoint are initially selected via Tendermint's weighted round-robin algorithm. A further custom check is implemented based on the checkpoint submission success. This allows the Polygon system to decouple with Tendermint proposer selection and provides Polygon with the abilities like selecting a proposer only when the checkpoint transaction on the Ethereum mainnet succeeds or submitting a checkpoint transaction for the blocks belonging to previous failed checkpoints.

Successfully submitting a checkpoint on Tendermint is a 2-phase commit process:

- A proposer, selected via the round-robin algorithm, sends a checkpoint with the proposer's address and the Merkle hash in the proposer field.
- All other proposers validate the data in the proposer field before adding the Merkle hash in their state.

The next proposer then sends an acknowledgment transaction to prove that the previous checkpoint transaction has succeeded on the Ethereum mainnet. Every validator set change is relayed by the validator nodes on Heimdall which is embedded onto the validator node. This allows Heimdall to remain in sync with the Polygon contract state on the Ethereum mainnet at all times.

The Polygon contract deployed on the Ethereum mainnet is considered to be the ultimate source of truth, and therefore all validation is done via querying the Ethereum mainnet contract.

id: derivatives title: Derivatives description: Delegation through validator shares keywords: - docs - polygon - matic - derivatives - delegation - shares slug: derivatives image: https://wiki.polygon.technology/img/polygon-logo.png

Polygon supports delegation via validator shares. By using this design, it is easier to distribute rewards and slash with scale on the Ethereum mainnet contracts without much computation.

Delegators delegate by purchasing shares of a finite pool from validators. Each validator has their own validator share token.

Let's call the fungible validator share tokens VATIC for Validator A. When a user delegates to Validator A, the user is issued VATIC based on the exchange rate of the MATIC-VATIC pair. As users accrue value, the exchange rate indicates that the user can withdraw more MATIC for each VATIC. When validators get slashed, users withdraw less MATIC for their VATIC.

Note that MATIC is the staking token. A delegator needs to have MATIC tokens to participate in the delegation.

Initially, Delegator D buys tokens from the Validator A specific pool when the exchange rate is 1 MATIC per 1 VATIC.

When a validator gets rewarded with more MATIC tokens, the new tokens are added to the pool.

Let's say with the current pool of 100 MATIC tokens, 10 MATIC rewards are added to the pool. Since the total supply of VATIC tokens did not change due to the rewards, the exchange rate becomes 1 MATIC per 0.9 VATIC. Now, Delegator D gets more MATIC for the same amount if shares.

The flow in the contract

`buyVoucher` : This function is attributed when performing a delegation process towards a validator. The delegation `_amount` is first transferred to `stakeManager` , which on confirmation mints delegation shares via `Mint` using the current `exchangeRate` .

The exchange rate is calculated as per the formula:

```
ExchangeRate = (totalDelegatedPower + delegatorRewardPool) / totalDelegatorShares
```

`sellVoucher` : This is function that is called when a delegator is unbonding from a validator. This function basically initiates the process of selling the vouchers bought during delegation. There is a withdrawal period that is taken into consideration before the delegators can `claim` their tokens.

`withdrawRewards` : As a delegator, you can claim your rewards by invoking the `withdrawRewards` function.

`reStake` : Restaking can work in two ways: a) delegator can buy more shares using `buyVoucher` or `reStake` rewards. You can restake by staking more tokens towarda a validator or you can restake your accumulated rewards as a delegator. Purpose of `reStaking` is that since delegator's validator has now more active stake, they will earn more rewards for that and so will the delegator.

`unStakeClaimTokens` : Once the withdrawal period is over, the delegators who sold their shares can claim their MATIC tokens.

`updateCommissionRate` : Updates the commission % for the validator. See also Validator Commission Operations.

`updateRewards` : When a validator gets rewards for submitting a checkpoint, this function is called for disbursements of rewards between the validator and delegators.

id: heimdall-chain title: Heimdall Chain description: Proof-of-stake verifier layer on the Polygon Network keywords: - docs - polygon - matic - heimdall - chain - verifier - layer - proof of stake slug: heimdall-chain image: https://wiki.polygon.technology/img/polygon-logo.png

Heimdall is the proof-of-stake verifier layer, which is responsible for checkpointing the representation of blocks to the Ethereum mainnet. Heimdall is based on Tendermint.

The staking contract on the Ethereum mainnet works in conjunction with the Heimdall node to act as the trustless stake management mechanism for the PoS engine, including selecting the validator set, updating validators, etc. Since staking is done in the contract on the Ethereum mainnet, Polygon does not rely only on validator honesty and instead inherits the Ethereum mainnet security.

Heimdall layer handles the aggregation of blocks produced by Bor into a Merkle tree and publishes the Merkle root periodically to the Ethereum mainnet. This periodic publishing is called *checkpointing*.

For every few blocks on Bor, a validator (on the Heimdall layer):

1. Validates all the blocks since the last checkpoint.
2. Creates a Merkle tree of the block hashes.
3. Publishes the Merkle root to the Ethereum mainnet.

Checkpoints are important for two reasons:

1. Providing finality on the root chain.
2. Providing proof of burn in withdrawal of assets.

An overview of the process:

- A subset of active validators from the pool is selected to act as block producers for a span. These block producers are responsible for creating blocks and broadcasting the created blocks on the the network.

- A checkpoint includes the Merkle root hash of all blocks created during any given interval. All nodes validate the Merkle root hash and attach their signature to it.
- A selected proposer from the validator set is responsible for collecting all signatures for a particular checkpoint and committing the checkpoint on the Ethereum mainnet.
- The responsibility of creating blocks and proposing checkpoints is variably dependent on a validator's stake ratio in the overall pool.

See also Heimdall architecture.

id: key-management title: Key Management description: Signer and owner keys management keywords: - docs - polygon - matic - key - key management - signer - owner slug: key-management image: https://wiki.polygon.technology/img/polygon-logo.png

Each validator uses two keys to manage validator related activities on Polygon:

- Signer key
- Owner key

Signer Key

The signer key is the address used to sign Heimdall blocks, checkpoints, and other signing related activities.

The signer address's private key must be located on the machine running the validator node for signing purposes.

The signer key cannot manage staking, rewards, or delegations.

The validator must keep ETH on the signer address on the Ethereum mainnet to send checkpoints.

Owner Key

The owner key is the address used to stake, restake, change the signer key, withdraw rewards and manage delegation related parameters on the Ethereum mainnet. The private key for the owner key must be secure at all costs.

All transactions through the owner key are performed on the Ethereum mainnet.

The signer key is kept on the node and is generally considered a **hot** wallet, whereas the owner key is supposed to kept very secure, is used infrequently, and is generally considered a **cold** wallet. The staked funds are controlled by the owner key.

This separation of responsibilities between the signer and the owner keys is done to ensure an efficient tradeoff between security and ease of use.

Both keys are Ethereum compatible addresses and work in the exactl the same manner.

Signer Change

See Change Your Signer Address.

id: proposer-bonus title: Proposer Bonus description: Additional incentive of being a validator keywords: - docs - polygon - matic - validate - proposer - bonus - incentive slug: proposer-bonus image: https://wiki.polygon.technology/img/polygon-logo.png

Proposer Bonus

In Polygon, there is an additional element of committing periodic checkpoints to the Ethereum mainnet. This is a major part of the validator responsibilities and they are incentivized to perform this activity. This constitutes a cost to the validator which is unique to a Layer 2 solution such as Polygon. We strive to accommodate this cost in the validator staking reward payout mechanism as a bonus to be paid to the proposer, who is responsible for committing the checkpoint. Rewards minus the bonus is to be shared among all stakers; proposer and signers, proportionally.

To avail the bonus completely, the proposer must include all signatures in the checkpoint. Because the protocol desires ⅔ +1 weight of the total stake, the checkpoint is accepted even with 80% votes. However, in this case, the proposer gets only 80% of the calculated bonus.

id: proposers-producers-selection title: Proposers & Producers Selection sidebar_label: Proposers & Producers description: Proposer & block producer selection on Polygon keywords: - docs - polygon - matic - proposers - block producers - selection slug: proposers-producers-selection image: https://wiki.polygon.technology/img/polygon-logo.png

import useBaseUrl from '@docusaurus/useBaseUrl';

Block Producers for the BOR layer, are a committee selected from the Validators pool on the basis of their stake which happens at regular intervals. These intervals are decided by the Validator's governance with regards to dynasty and network.

The ratio of stake specifies the probability to be selected as a member of block producers committee.

Selection Process

Let's suppose we have 3 validators in pool — Alice, Bill, and Clara:

- Alice is staking 100 MATIC tokens.
- Bill is staking 40 MATIC tokens.
- Clara is staking 40 MATIC tokens.

Validators are given slots according to the stake.

Because Alice has 100 MATIC tokens staked, and the per slot cost is 10 MATIC tokens as maintained by validator's governance, Alice gets 5 slots in total. Similarly, Bill and Clara get 2 slots in total.

The Alice, Bill and Clara validators are given the following slots:

- [A, A, A, A, A, B, B, C, C]

Polygon then shuffles the array of the Alice, Bill and Clara slots by using the Ethereum block hashes as seed.

The result of the shuffle is the following array of slots:

- [A, B, A, A, C, B, A, A, C]

Now depending on the total block producer count as maintained by validator's governance, Polygon uses the validators from the top — for example. for a set of 5 producers the array of slots is [A, B, A, A, C].

The producer set for the next span is defined as [A: 3, B:1, C:1].

Using the resulting validator set and Tendermint's proposer selection algorithm, Polygon selects a producer for every sprint on Bor.

<img src={useBaseUrl("img/validators/producer-proposer.png")} />

Legend:

- Dynasty: Time between the end of the last auction and start time of the next auction.
- Sprint: Time interval for which the block producers committee is selected.
- Span: Number of blocks produced by a single producer.

---

id: staking title: Staking sidebar_label: Staking description: Stake, unstake, and restake as a validator keywords: - docs - matic - polygon - staking - unstake - restake - validator slug: staking image: https://wiki.polygon.technology/img/polygon-logo.png

For Polygon Network, any participant can be qualified to become a Polygon's validator by running a full node to earn rewards and collect transaction fees. To ensure the good participation by validators, they lock up some of their MATIC tokens as a stake in the ecosystem.

Validators in Polygon Network are selected via an on-chain auction process which happens at regular intervals.

A validator has two addresses an owner address and a signer address. The staking is done with the owner address.

See also Key Management.

Stake

:::note

Currently there is limited space to accept new validators.

A new validator can only join the active set when a currently active validator unbonds.

:::

To join the validator set, you must stake your MATIC tokens. See Validator Staking Operations.

Unstake

Unstaking allows the validator to be out of active pool of validators.

To ensure good participation, the validator stake is locked for 80 checkpoints.

Restake

Validators can add more MATIC tokens to their stake:

- To earn more rewards.
- To maintain the position in the validator set.

---

id: state-sync-mechanism title: State Sync Mechanism description: State sync mechanism to natively read Ethereum data keywords: - docs - matic - polygon - state sync - mechanism slug: state-sync-mechanism image: https://wiki.polygon.technology/img/polygon-logo.png

Validators on the Heimdall layer pick up the StateSynced event and pass the event on to the Bor layer. See also Polygon Architecture.

The **receiver contract** inherits IStateReceiver, and custom logic sits inside the onStateReceive function.

The latest version, Heimdall v0.3.4, contains a few enhancements such as:

1. Restricting data size in state sync txs to:
    - **30Kb** when represented in **bytes**
    - **60Kb** when represented as **string**.
2. Increasing the **delay time** between the contract events of different validators to ensure that the mempool doesn't get filled very quickly in case of a burst of events which can hamper the progress of the chain.

The following example shows how the data size is restricted:

```
Data - "abcd1234"
Length in string format - 8
Hex Byte representation - [171 205 18 52]
Length in byte format - 4
```

Requirements for the users

Things required from dapps/users to work with state-sync are:

1. Call the syncState function.
2. The `syncState` function emits an event called `StateSynced(uint256 indexed id, address indexed contractAddress, bytes data);`
3. All the validators on the Heimdall chain receive the `StateSynced` event. Any validator that wishes to get the transaction fee for the state sync sends the transaction to Heimdall.
4. Once the `state-sync` transaction on Heimdall is included in a block, it is added to the pending state-sync list.
5. After every sprint on Bor, the Bor node fetches the pending state-sync events from Heimdall via an API call.
6. The receiver contract inherits the `IStateReceiver` interface, and custom logic of decoding the data bytes and performing any action sits inside the onStateReceive function.

---

id: transaction-fees title: Transaction Fees description: Distribution of fees among all validators on Heimdall keywords: - docs - polygon - matic - validator - transaction - fees slug: transaction-fees image: https://wiki.polygon.technology/img/polygon-logo.png

Each block producer on the Bor layer is given a certain percentage of the transaction fees collected in each block.

The selection of producers for any given span is dependent on the validator's ratio in the overall stake. The remaining transaction fees flow through the same funnel as the rewards which get shared among all validators working at the Heimdall layer.

id: responsibilities title: Responsibilities description: The responsibilities of being a validator on the Polygon Network keywords: - docs - matic - polygon - validate - validator - responsibilities slug: responsibilities image: https://wiki.polygon.technology/img/polygon-logo.png

:::tip Stay in the know

Keep up with the latest node and validator updates from the Polygon team and the community by subscribing to the Polygon notification groups.

:::

A blockchain validator is someone who is responsible for validating transactions within a blockchain. On the Polygon Network, any participant can be qualified to become a Polygon's validator by running a **Validator Node (Sentry + Validator)** to earn rewards and collect transaction fees. To ensure the good participation by validators, they lock up at least 1 MATIC token as a stake in the ecosystem.

:::info

Currently, there is a limit of 100 active validators at a time. For a detailed description on what a validator is, see Validator.

Also, after the PIP4 governance proposal is implemented on the contract-level, the minimum staking amount will increase to 10,000 MATIC.

:::

Any validator on the Polygon Network has the following responsibilities:

- Technical node operations (done automatically by the nodes)
- Operations
    - Maintain high uptime
    - Check node-related services and processes daily
    - Run node monitoring
    - Keep ETH balance (between 0.5 to 1) on the signer address
- Delegation
    - Be open to delegation
    - Communicate commission rates
- Communication
    - Communicate issues
    - Provide feedback and suggestions
- Earn staking rewards for validate blocks on the blockchain

Technical node operations

The following technical node operations are **done automatically by the nodes:**

- Block producer selection:
  - Select a subset of validators for the block producer set for each span
  - For each span, select the block producer set again on Heimdall and transmit the selection information to Bor periodically.
- Validating blocks on Bor:
  - For a set of Bor blocks, each validator independently reads block data for these blocks and validates the data on Heimdall.
- Checkpoint submission:
  - A proposer is chosen among the validators for each Heimdall block. The checkpoint proposer creates the checkpoint of Bor block data, validates, and broadcasts the signed transaction for other validators to consent to.
  - If more than 2/3 of the active validators reach consensus on the checkpoint, the checkpoint is submitted to the Ethereum mainnet.
- Sync changes to Polygon staking contracts on Ethereum:
  - Continuing from the checkpoint submission step, since this is an external network call, the checkpoint transaction on Ethereum may or may not be confirmed, or may be pending due to Ethereum congestion issues.
  - In this case, there is an `ack/no-ack` process that is followed to ensure that the next checkpoint contains a snapshot of the previous Bor blocks as well. For example, if checkpoint 1 is for Bor blocks 1-256, and it failed for some reason, the next checkpoint 2 will be for Bor blocks 1-512. See also Heimdall architecture: Checkpoint.
- State sync from the Ethereum mainnet to Bor:
  - Contract state can be moved between Ethereum and Polygon, specifically through Bor:
  - A DApp contract on Ethereum calls a function on a special Polygon contract on Ethereum.
  - The corresponding event is relayed to Heimdall and then Bor.
  - A state-sync transaction gets called on a Polygon smart contract and the DApp can get the value on Bor via a function call on Bor itself.
  - A similar mechanism is in place for sending state from Polygon to Ethereum. See also State Sync Mechanism.

Operations

Maintain high uptime

The node uptime on the Polygon Network is based on the number of checkpoint transactions that the validator node has signed.

Approximately every 34 minutes a proposer submits a checkpoint transaction to the Ethereum mainnet. The checkpoint transaction must be signed by every validator on the Polygon Network. **Failure to sign a checkpoint transaction results in the decrease of your validator node performance**.

The process of signing the checkpoint transactions is automated. To ensure your validator node is signing all valid checkpoint transactions, you must maintain and monitor your node health.

Check node services and processes daily

You must check daily the services and processes associated with Heimdall and Bor. Also, pruning of the nodes should be done regularly to reduce disk usage.

Run node monitoring

You must run either:

- Grafana Dashboards provided by Polygon. See GitHub repository: Matic-Jagar setup
- Or, use your own monitoring tools for the validator and sentry nodes
- Ethereum endpoint used on nodes should be monitored to ensure the node is within the request limits

Keep an ETH balance

You must maintain an adequate amount of ETH (should be always around the threshold value i.e., 0.5 to 1) on your validator signer address on the Ethereum Mainnet.

You need ETH to:

- Sign the proposed checkpoint transactions on the Ethereum Mainnet.
- Propose and send checkpoint transactions on the Ethereum Mainnet.

Not maintaining an adequate amount of ETH on the signer address will result in:

- Delays in the checkpoint submission. Note that transaction gas prices on the Ethereum network may fluctuate and spike.
- Delays in the finality of transactions included in the checkpoints.
- Delays in subsequent checkpoint transactions.

Delegation

Be open for delegation

All validators must be open for delegation from the community. Each validator has the choice of setting their own commission rate. There is no upper limit to the commission rate.

Communicate commission rates

It is the moral duty of the validators to communicate the commission rates and the commission rate changes to the community. The preferred platforms to communicate the commission rates are:

- Discord
- Forum

Communication

Communicate issues

Communicating issues as early as possible ensures that the community and the Polygon team can rectify the problems as soon as possible. The preferred platforms to communicate the commission rates are:

- Discord
- Forum
- GitHub

Provide feedback and suggestions

At Polygon, we value your feedback and suggestions on any aspect of the validator ecosystem. Forum is the preferred platform to provide feedback and suggestions.

id: rewards title: Rewards sidebar_label: Rewards description: Learn about the Polygon Network staking incentives. keywords: - docs - matic - polygon - rewards - staking - incentives image: https://wiki.polygon.technology/img/polygon-logo.png

In Polygon, validators stake their MATIC tokens as collateral to work for the security of the network, and in exchange for their service, earn rewards.

To leverage Polygon's economics, you should either become a validator or a delegator.

To be a validator, you need to **run a full validator** node and stake MATIC. See Validate.

Also check the Validator Responsibilities page.

To be a delegator, you only need to **delegate MATIC to a validator**. See Delegate.

What is the incentive?

Polygon allocates 12% of its total supply of 10 billion tokens to fund the staking rewards. This is to ensure that the network is seeded well enough until transaction fees gain traction. These rewards are primarily meant to jump-start the network, while the protocol in the long run is intended to sustain itself on the basis of transaction fees.

Validator Rewards = Staking Rewards + Transaction Fees

This is allocated in a way to ensure gradual decoupling of staking rewards from being the dominant component of the validator rewards.

Year	Target Stake (30% of circulating supply)	Reward Rate for 30% Bonding	Reward Pool
First	1,977,909,431	20%	312,917,369
Second	2,556,580,023	12%	275,625,675
Third	2,890,642,855	9%	246,933,140
Fourth	2,951,934,048	7%	204,303,976
Fifth	2,996,518,749	5%	148,615,670 + **11,604,170**

Below is a sample snapshot of the expected annual rewards for the first 5 years considering staked supply ranging from 5% to 40% at 5% interval

% of circulating supply staked	5%	10%	15%	20%	25%	30%	35%	40%
Annual reward for year								
First	120%	60%	40%	30%	24%	20%	17.14%	15%
Second	72%	36%	24%	18%	14.4%	12%	10.29%	9%
Third	54%	27%	18%	13.5%	10.8%	9%	7.71%	6.75%
Fourth	42%	21%	14%	10.5%	8.4%	7%	6%	5.25%
Fifth	30%	15%	10%	7.5%	6%	5%	4.29%	3.75%

Who gets the incentives?

Stakers running validator nodes and stakers delegating their tokens toward a validator that they prefer.

Validators have the option to charge a commission on the reward earned by delegators.

The funds belonging to all stakers are locked in a contract deployed on the Ethereum mainnet.

No validator holds custody over delegator tokens.

Staking rewards

The yearly incentive is absolute — irrespective of the overall stake or the target bonding rate in the network, the incentive amount is given out as a reward to all signers periodically.

In Polygon, there is an additional element of committing periodic checkpoints to the Ethereum mainnet. This is a major part of the validator responsibilities and they are incentivized to perform this activity. This constitutes a cost to the validator which is unique to a Layer 2 solution such as Polygon. We strive to accommodate this cost in the validator staking reward payout mechanism as a bonus to be paid to the proposer, who is responsible for committing the checkpoint. Rewards minus the bonus is to be shared among all stakers, proposer and signers, proportionally.

Encouraging the proposer to include all signatures

To avail the bonus completely, the proposer must include all signatures in the checkpoint. Because the protocol desires ⅔ +1 weight of the total stake, the checkpoint is accepted even with 80% votes. However, in this case, the proposer gets only 80% of the calculated bonus.

Transaction fees

Each block producer at Bor is given a certain percentage of the transaction fees collected in each block. The selection of producers for any given span is also dependent on the validator's ratio in the overall stake. The remaining transaction fees flow through the same funnel as the rewards which get shared among all validators working at the Heimdall layer.

id: dagger title: Dagger sidebar_label: Dagger - Single App description: Build your next blockchain app on Polygon keywords: - docs - matic - polygon - dagger image: https://wiki.polygon.technology/img/polygon-logo.png

import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem';

:::caution Content disclaimer

Please view the third-party content disclaimer here.

:::

Dagger is the best way to get realtime updates from Ethereum Blockchain. It provides a way for your DApps and Backend system to get Ethereum blockchain events i.e. transactions, token transfers, receipts and logs in realtime over websocket or socket.

We maintain infrastructure for reliable and scalable realtime events. `@maticnetwork/dagger` is consumer library for Dagger project written in NodeJS. It uses Dagger server to get realtime updates from Ethereum Network.

Installation

```
# Using Yarn
yarn add @maticnetwork/dagger

# Using NPM
npm install @maticnetwork/dagger --save
```

Network

Ethereum Network

Mainnet

```
Websocket: wss://mainnet.dagger.matic.network
Socket: mqtts://mainnet.dagger.matic.network (You can also use `ssl://` protocol)
```

Kovan

```
Websocket: wss://kovan.dagger.matic.network
Socket: mqtts://kovan.dagger.matic.network (You can also use `ssl://` protocol)
```

Ropsten

```
Websocket: wss://ropsten.dagger.matic.network
Socket: mqtts://ropsten.dagger.matic.network (You can also use `ssl://` protocol)
```

Goerli

```
Websocket: wss://goerli.dagger.matic.network
Socket: mqtts://goerli.dagger.matic.network (You can also use `ssl://` protocol)
```

Matic Network

Mainnet

```
Websocket: wss://matic-mainnet.dagger.matic.network
Socket: mqtts://matic-mainnet.dagger.matic.network (You can also use `ssl://` protocol)
```

Mumbai Testnet

```
Websocket: wss://mumbai-dagger.matic.today
Socket: mqtts://mumbai-dagger.matic.today (You can also use `ssl://` protocol)
```

Example

- Lets first create a *npm* project.

```
npm init -y
touch index.js
```

- Now we can put following code snippet in `index.js`.

```
const Dagger = require('@maticnetwork/dagger')

// connect to correct dagger server, for receiving network specific events
//
// you can also use socket based connection
const dagger = new Dagger("wss://mainnet.dagger.matic.network")

// get new block as soon as it gets created
dagger.on('latest:block.number', result => {
  console.log(`New block created: ${result}`)
})
```

- Run `index.js` & you'll start receiving block number as soon as new block gets created.

```
node index.js
```

API

new Dagger(url)

Create dagger object

- `url` is dagger server's address. Check network section for all available url values.

Example:

```
const dagger = new Dagger(<url>)
```

dagger.on(event, fn)

Subscribe to a topic

- `event` is a `String` topic to subscribe to. `event` wildcard characters are supported (`+` - for single level and `#` - for multi level)
- `fn` - `function (data, removed)` fn will be executed when event occurred:
  - `data` data from event
  - `removed` flag saying if data is removed from blockchain due to re-organization.

Example:

```
dagger.on('latest:block.number', (res, flag) => { console.log(res, flag) })
```

dagger.once(event, fn)

Same as on but will be fired only once.

Example:

```
dagger.once('latest:block.number', (res, flag) => { console.log(res, flag) })
```

dagger.off(event, fn)

Unsubscribe from a topic

- `event` is a `String` topic to unsubscribe from
- `fn` - `function (data, removed)`

Example:

```
dagger.off('latest:block.number', (res, flag) => { console.log(res, flag) })
```

dagger.of(room)

Create room out of dagger. `room` has to be one out of two values

- `latest`
- `confirmed`

`room` object has following methods:

- `on` same as dagger `on`
- `once` same as dagger `once`
- `off` same as dagger `off`

```
const latestRoom = dagger.of('latest')
const confirmedRoom = dagger.of('confirmed')
```

dagger.end([force])

Close the dagger, accepts the following options:

- `force` : passing it to true will close the dagger right away. This parameter is optional.

```
dagger.end({force: true}) // immediate closing
```

dagger.contract(web3Contract)

Creates web3 contract wrapper to support Dagger.

- First create a web3 contract object.

```
// web3 contract
const web3Contract = new web3.eth.Contract(abi, address)
```

- Now we'll create a dagger contract wrapper on it.

```
// dagger contract
const contract = dagger.contract(web3Contract)
```

- Time to filter out contract events

```
const filter = contract.events.Transfer({
  filter: { from: "0x123456..." },
  room: "latest"
})
```

- Watching contract events

```
// watch
filter.watch((data, removed) => { console.log(data, removed) })

// or watch only once
filter.watchOnce((data, removed) => { console.log(data, removed) })
```

- Stopping event watching

```
// stop watching
filter.stopWatching();
```

Events

Every event has a room ∈ { `latest` , `confirmed` }.

- `latest` : Events are fired immediately after block included in chain.
- `confirmed` : Events are fired after 12 confirmations.

If you want to show updates on UI in your DApp, use `latest` events. It will help to make UI/UX better and user friendly.

Use `confirmed` events for irreversible tasks from server or on UI. Like sending email, notifications or allow user to do subsequent task on UI after one transaction gets confirmed.

Network Events

Ethereum event	When?	`removed` flag
block	For every new block created	Yes
block.number	For every new block number created	
block.hash	For every new block hash created	Yes
block/`number`	When particular block in future included in chain	Yes
addr/`address`/tx	On every new transaction for `address`	Yes
addr/`address`/tx/out	On every new outgoing transaction for `address`	Yes
addr/`address`/tx/in	On every new incoming transaction for `address`	Yes
tx/`txId`	When given `txId` included in block	Yes
tx/`txId`/success	When tx status is success (included in block) for `txId`	Yes
tx/`txId`/fail	When tx fails (included in block) for `txId`	Yes
tx/`txId`/receipt	When receipt is generated (included in block) for `txId`	Yes
addr/`contractAddress`/deployed	When new `contractAddress` included in block	Yes
log/`contractAddress`	When new log generated for `contractAddress`	Yes
log/`contractAddress`/filter/`topic1`/`topic2`	When new log with `topic1` and `topic2` generated for `contractAddress`	Yes

Dagger Events

Dagger Event	When?	args
connection.status	When connection status changes	value: Boolean

Every event has to start with room:

block

For every new block

<Tabs defaultValue="latest" values={[{ label: 'latest', value: 'latest', }, { label: 'confirmed', value: 'confirmed', },] }>

```
dagger.on("latest:block", result => {
  console.log("Current block : ", result)
})
```

```
dagger.on("confirmed:block", result => {
  console.log("Confirmed block : ", result)
})
```

block.number

For every new block number

<Tabs defaultValue="latest" values={[{ label: 'latest', value: 'latest', }, { label: 'confirmed', value: 'confirmed', },] }>

```
dagger.on("latest:block.number", result => {
  console.log("Current block number : ", result)
})
```

```
dagger.on("confirmed:block.number", result => {
  console.log("Confirmed block number : ", result)
})
```

block.hash

For every new block hash

<Tabs defaultValue="latest" values={[{ label: 'latest', value: 'latest', }, { label: 'confirmed', value: 'confirmed', },] }>

```
dagger.on("latest:block.hash", result => {
  console.log("Current block hash : ", result)
})
```

```
dagger.on("confirmed:block.hash", result => {
  console.log("Confirmed block hash : ", result)
})
```

block/{number}

When particular block **X**, in future included in chain

<Tabs defaultValue="latest" values={[{ label: 'latest', value: 'latest', }, { label: 'confirmed', value: 'confirmed', },] }>

```
dagger.on("latest:block/X", result => {
  console.log("Included in chain : ", result)
})
```

```
dagger.on("confirmed:block/X", result => {
  console.log("Included in chain : ", result)
})
```

addr/{address}/tx

On every new transaction for `address`

<Tabs defaultValue="latest" values={[{ label: 'latest', value: 'latest', }, { label: 'confirmed', value: 'confirmed', },] }>

```
dagger.on("latest:addr/{address}/tx", result => {
  console.log("New Transaction : ", result)
})
```

```
dagger.on("confirmed:addr/{address}/tx", result => {
  console.log("New Transaction : ", result)
})
```

addr/{address}/tx/{dir}

`dir` is transaction direction ∈ { `in` , `out` }. `address` can be omitted to receive notification for any address.

<Tabs defaultValue="in" values={[{ label: 'incoming', value: 'in', }, { label: 'outgoing', value: 'out', }, { label: 'wild card', value: 'all', },] }>

On every new incoming transaction for `address`

<Tabs defaultValue="latest" values={[{ label: 'latest', value: 'latest', }, { label: 'confirmed', value: 'confirmed', },] }>

```
dagger.on("latest:addr/{address}/tx/in", result => {
  console.log("New Incoming Transaction : ", result)
})
```

```
dagger.on("confirmed:addr/{address}/tx/in", result => {
  console.log("New Incoming Transaction : ", result)
})
```

On every new outgoing transaction for `address`

<Tabs defaultValue="latest" values={[{ label: 'latest', value: 'latest', }, { label: 'confirmed', value: 'confirmed', },] }>

```
dagger.on("latest:addr/{address}/tx/out", result => {
  console.log("New Outgoing Transaction : ", result)
})
```

```
dagger.on("confirmed:addr/{address}/tx/out", result => {
  console.log("New Outgoing Transaction : ", result)
})
```

Using wildcard notation in place of `address` , to get notified for all incoming & outgooing transactions.

<Tabs defaultValue="latest" values={[{ label: 'latest', value: 'latest', }, { label: 'confirmed', value: 'confirmed', },] }>

```
dagger.on("latest:addr/+/tx/in", result => {
  console.log("New Incoming Transaction : ", result)
})
```

```
dagger.on("confirmed:addr/+/tx/in", result => {
  console.log("New Incoming Transaction : ", result)
```

```
})
```

tx/{txId}/{status}

`status` is `txId` 's status ∈ { `success` , `fail` , `receipt` }. It can be kept empty too i.e. resulting into `tx/{txId}` , triggered when `txId` gets included in block.

<Tabs defaultValue="any" values={[{ label: 'any', value: 'any', }, { label: 'success', value: 'success', }, { label: 'fail', value: 'fail', }, { label: 'receipt', value: 'receipt', },] }>

When given `txId` included in block

<Tabs defaultValue="latest" values={[{ label: 'latest', value: 'latest', }, { label: 'confirmed', value: 'confirmed', },] }>

```
dagger.on("latest:tx/{txId}", result => { console.log(result) })
```

```
dagger.on("confirmed:tx/{txId}", result => { console.log(result) })
```

When tx status is success (included in block) for `txId`

<Tabs defaultValue="latest" values={[{ label: 'latest', value: 'latest', }, { label: 'confirmed', value: 'confirmed', },] }>

```
dagger.on("latest:tx/{txId}/success", result => { console.log(result) })
```

```
dagger.on("confirmed:tx/{txId}/success", result => { console.log(result) })
```

When tx fails (included in block) for `txId`

<Tabs defaultValue="latest" values={[{ label: 'latest', value: 'latest', }, { label: 'confirmed', value: 'confirmed', },] }>

```
dagger.on("latest:tx/{txId}/fail", result => { console.log(result) })
```

```
dagger.on("confirmed:tx/{txId}/fail", result => { console.log(result) })
```

When receipt is generated (included in block) for `txId`

<Tabs defaultValue="latest" values={[{ label: 'latest', value: 'latest', }, { label: 'confirmed', value: 'confirmed', },] }>

```
dagger.on("latest:tx/{txId}/receipt", result => { console.log(result) })
```

```
dagger.on("confirmed:tx/{txId}/receipt", result => { console.log(result) })
```

log/{contractAddress}

When log generated for `contractAddress`

<Tabs defaultValue="latest" values={[{ label: 'latest', value: 'latest', }, { label: 'confirmed', value: 'confirmed', },] }>

```
dagger.on("latest:log/{contractAddress}", result => {
  console.log("New Log : ", result)
})
```

```
dagger.on("confirmed:log/{contractAddress}", result => {
  console.log("New Log : ", result)
})
```

log/{contractAddress}/filter/{topic0}/{topic1}/{topic2}

When new log with `topic0` , `topic1` & `topic2` generated for `contractAddress`

```
// Triggers when 1 GNT (Golem token) get transferred to Golem multisig wallet
dagger.on('latest:log/0xa74476443119a942de498590fe1f2454d7d4ac0d/filter/0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4d
console.log)

// Triggers when any amount of GNT (Golem token) get sent from Golem multisig wallet
dagger.on('latest:log/0xa74476443119a942de498590fe1f2454d7d4ac0d/filter/0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4d
...)

// Listen for every Golem token transfer (notice `#` at the end)
dagger.on('latest:log/0xa74476443119a942de498590fe1f2454d7d4ac0d/filter/0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4d
...)
```

> *Event names are case-sensitive.* `address` , `txId` *and* `topics` *must be in lowercase.*

> *Note: You can use wildcard for events too. There are two type of wildcards: `+` (for single) and `#` (for multiple). Use with caution as it will fetch more data then you need, and can bombard with data to your DApp.*

Test Dagger Server

This library consists `woodendagger` executable which is test dagger server on your local machine. So you can test with TestRPC.

Please do not use `woodendagger` in production. It's only for development purpose. It doesn't support `removed` flag.

```
$ woodendagger --url=https://mainnet.infura.io # or http://localhost:8545 for local json-rpc

# If you want to start dagger server on different ports,
# sockport: socket port for backend connection over TCP
# wsport: websocket port for frontend connection over websocket
$ woodendagger --url=http://localhost:8545 --sockport=1883 --wsport=1884

# To connect from dagger:
const dagger = new Dagger('mqtt://localhost:1883')
```

Support

If you have any queries, feedback or feature requests, feel free to reach out to us on Telegram

License

MIT

id: dapp-fauna-polygon-react sidebar_label: dApps Data Tutorial title: Data Tutorial for Decentralized Applications description: Build dapps using Fauna, Polygon, and React keywords: - docs - matic - dapp - fauna - polygon - data - security - react image: https://wiki.polygon.technology/img/polygon-logo.png slug: dapp-fauna-polygon-react

In this tutorial, we'll be building a simple allowlisting app that utilizes **React** for the UI and functionality, **Polygon PoS** for transactions, and **Fauna** for storing private data from transactions that we may not necessarily want to surface publicly in the blockchain. **Allowlisting**, also known as **whitelisting**, is a concept that is very common in the decentralized world - when signing up for a allowlist, you typically gain access to special privileges, such as being able to be the first to purchase digital assets.

img

GIF of app workflow - user fills out form, MetaMask window pops up, and then once the transaction is submitted and confirmed, it surfaces a success message.

The application will take in a first name, last name, and a wallet address. Typically, a lot of allowlists take in a wallet address only, but we'll be using the first and last name as a way to pass in some additional private information within the transaction. The flow of data will be as illustrated in the diagram:

img User submits their first name, last name, and wallet address successfully through a form. The new database entry is submitted to Fauna that has a

randomly generated UUID, first name, last name, and wallet address. Net transaction is submitted to Polygon that passes in first name, last name, and wallet address as part of the transaction details, but each property equals the UUID generated for the Fauna database.

What is Fauna?

Fauna is a data API that offers a serverless database experience. With in-depth documentation and a web-native GraphQL interface, it allows developers to quickly get started with storing their application data without needing to sacrifice things such as flexibility, scale, and performance.

In this tutorial, we'll be using Fauna as a method of storing private transaction data we may not want to surface on the blockchain explicitly. We'll use UUIDs populated in the private transaction data to map information from a blockchain transaction to a specific entry with the actual private transaction details in the Fauna database.

Getting started

In order to get started with building, follow these steps:

- Create a Fauna Account — you can sign up here.
- Create a MetaMask Wallet with Polygon's Test Network (Mumbai-Testnet) configured.
  - First, get set up with MetaMask here. Make sure you save your Secret Recovery Phrase.
  - Then, configure the Mumbai-Testnet on your MetaMask by following the instructions here.
- Once you have MetaMask set up and configured for the Mumbai-Testnet, you'll need some MATIC to add to your wallet
  - MATIC is the native cryptocurrency of the Polygon network and it's used to pay network fees, for staking, and also for governance to the Polygon blockchain (MATIC holders can vote on Polygon changes). In the context of this project, you will need MATIC to pay gas fees that are charged for each transaction. You can learn more about MATIC here.
  - To get started, use the MATIC faucet to get free MATIC sent to your wallet (note: this MATIC will only be available on the Mumbai-Testnet and is only for development purposes). Once on the webpage, make sure the following options are selected:

Network: Mumbai, Select Token: MATIC Token, Wallet Address: Put your wallet address here

- To find out what your wallet address is, you can pull it from Metamask:

Go into MetaMask from your browser. At the top under your Account Name, you'll see a long string. That's your account ID. Copy that ID by clicking on it.

:::info

This can take a while and you may have to try a few times before the first transaction comes through your wallet. Each successful transaction from the MATIC faucet will give you 0.5 MATIC.

Since testing the application requires spending MATIC as gas fees (a fluctuating fee that you must pay to perform transactions), we recommend doing this transaction ~12 times so you have about 6 MATIC in your wallet, which should be enough for testing the app in development.

To track your wallet's transactions, you can visit https://mumbai.polygonscan.com/address/[wallet address].

:::

- Now, install Truffle — this is a suite of tools designed for developing Ethereum applications.
  - To see if you already have Truffle installed, you can run `truffle version` - if you get back a response that looks like this, you already have it installed:

    ```
    $ truffle version
    Truffle v5.4.29 (core: 5.4.29)
    Solidity v0.5.16 (solc-js)
    Node v13.8.0
    Web3.js v1.5.3
    ```

  - To install Truffle, run `npm -g truffle`.
- Also, if you don't already have it, install the tool to spin up a sample React app with `npm i create-react-app`. Learn more about React here.

Start building

If you're someone that prefers walking through code independently, the GitHub repo is available here. To get started with the GitHub repo, download or clone it and reference the README to get set up.

Setup

1. Create a new directory for your project in your location of choice - we called ours `polygon-fauna-app`.
2. Once the directory is created, in your command line, `cd` into the folder (ex: `cd polygon-fauna-app`).
3. Once inside the directory, it's time to set up the app. Truffle makes it easier for us to build a decentralized application with a React front-end. To set up your initial project, run `truffle unbox react`.
   1. Once the command has run, if you were to run `ls` in your current directory, you'd notice a variety of different subdirectories that weren't there before. The most important ones we'll be focusing on are:
      1. `client` - this is where your application, its primary functionality, and its front-end will live. Within client, the structure looks exactly like a regular React project.
      2. `contracts` - this directory is where your smart contracts will live. Smart contracts are programs that will execute specified functionality within an account on the blockchain when the functionality is called. You might notice that the files in this directory end in `.sol` - this is because they are built with Solidity, which is an object-oriented, high-level language for implementing smart contracts.
      3. `migrations` - within this directory, there are preconfigured scripts that will allow you to deploy the smart contracts in `contracts` on the specified network, which will end up being the Mumbai-Testnet network.

:::note

```
If you use an IDE to code, now would be the perfect time to open your project in the IDE! We're going to start editing some files pretty soon.
```

:::

Smart contract configuration and functionality

1. Currently, the project is pointed towards deploying on the Ethereum blockchain directory. We're going to need to update this to point to the Mumbai-Testnet that we want to work off of.
   - To do this, first we'll need to run these commands in your command line to install some necessary packages:

     `npm install @truffle/hdwallet-provider`

     and

     `npm install dotenv`

   - Then, create a file called `.env` in the directory you're currently in. Within `.env`, do the following:

- Create an environment variable called `MNEMONIC` and make it equal to the Secret Recovery Phrase for your MetaMask wallet - if you didn't write it down, you can follow this guide to reveal it once again.
- Create an environment variable called `RPC_APP_ID` and

Your `.env` should look something like this:

```
MNEMONIC=your secret phrase should go here
```

- Finally, replace the truffle-config.js file that was generated through Truffle with the following (note: it should be in the same directory as the client, contracts, and migrations folders):

```
const HDWalletProvider = require("@truffle/hdwallet-provider");
const path = require("path");
require("dotenv").config(); // Load .env file

module.exports = {
  // See <http://truffleframework.com/docs/advanced/configuration>
  // to customize your Truffle configuration!
  contracts_build_directory: path.join(__dirname, "client/src/contracts"),
  networks: {
    develop: {
      port: 8545,
    },
    matic: {
      provider: () =>
        new HDWalletProvider(
          process.env.MNEMONIC,
          `https://matic-mumbai.chainstacklabs.com`
        ),
      network_id: 80001,
      confirmations: 2,
      timeoutBlocks: 200,
      skipDryRun: true,
      gas: 6000000,
      gasPrice: 10000000000,
    },
  },
};
```

2. Once you've set up your configuration, test it and make sure it points to your account.

- To test this, migrate your current smart contracts (Truffle has auto-populated some for you) by running `truffle migrate --network matic`. If it's not your first time running this `deploy` command for this project, you'll want to run `truffle migrate --network matic --reset` so it runs a fresh copy of migrations and pulls the most recent code updates.

:::info Facing Errors?

If you run into any issues, try the following things:

- Make sure your `MNEMONIC` environment variable is defined in a `.env` file in your root directory.
- If you are running into continuous errors and you've checked the above, try other RPC URLs in place of the matic-mubai.chainstacklabs.com URL in `truffle-config.js`. A list of additional URLs for the Mumbai-Testnet can be found on this page under the **Mumbai-Testnet** section.

:::

- Then, test being able to connect to your new application by running the front-end. Truffle added some pre-configured logic that will allow you to test this connection. You can test it by running the following from your main directory:

```
`cd client`
`npm run start`
```

:::note

You will only be able to start your application inside the `client` folder.

:::

- Your browser will launch a development instance on http://localhost:3000 that will then prompt Metamask. It may ask you to re-auth your password. Finally, MetaMask will attempt to connect your account on the testnet to your application instance, which will cost you some MATIC in gas fees. You'll see a window like this:

img

- To go through with the transaction, click "Confirm". The MetaMask window will disappear and you should get a small "Transaction Confirmed" notification in your browser. To verify that your wallet has been connected successfully, you should see this page:

img

Note that the stored value at the bottom of the page in this screenshot is the value 5. The original value is set to 0 when the page is initially loaded prior to authenticating your wallet. It will update to 5 once it is successfully connected.

3. Now, we can get to work on updating your smart contract functionality.

- To do this, we're going to clean up the `contracts` directory in the root project directory. You can access it in your command line by running `cd ../contracts` from the `client` directory.
- We'll first delete the default `SimpleStorage.sol` file in the `contracts` directory.
- Then, we'll add a new smart contract for the specific allowlisting functionality. Within `contracts`, create a new file called `Allowlist.sol` and paste in the following:

```
pragma solidity ^0.5.0;

contract Allowlist {
    // all are going to be set to uuid value in smart contract, so declare as same type
  struct allowlister {
    string f_name;
    string l_name;
    string wallet_address;
  }

  allowlister[] allowlisters; // array of all allowlisters

  function _createAllowlister (string memory _uuid) public {
    allowlisters.push(allowlister({
      f_name: _uuid,
      l_name: _uuid,
      wallet_address: _uuid
    })) -1;
  }
}
```

Let's break this code down a bit:

- The smart contract, `Allowlist`, contained an outline of what an allowlister (someone who submits an entry to the Allowlist) will look like. We know that we'll be taking in a first name, last name, and wallet address in the form, so we created a `struct` representing an allowlister with those properties. When we store the items in the contract, they'll each map to that submission's UUID, meaning that each variable type should align with whatever variable type that UUID will be.
    - Since the UUID will be a string that exceeds the length of other kinds of variable types in Solidity, we'll be assigning each property the type `string`, which accepts longer lengths.
- We've also initialized an array of `allowlisters` that we can add to as users submit information in the form.
- Finally, we've created a function, `_createAllowlister`, that will be what we call when users submit the form. The function takes in the generated `_uuid` string for that entry (the underscore in the variable name is to differentiate the variable as a function parameter). Within the function, we will push a new `allowlister` instance to the `allowlisters` array - each property in the instance will equal the `_uuid` that is passed in.
- Then, go into the migrations folder in the root directory. We're going to hop into the 2_deploy_contracts.js file and update the functionality so it deploys the new contract

```
var Allowlist = artifacts.require("./Allowlist.sol");

module.exports = function(deployer) {
  deployer.deploy(Allowlist);
};
```

- Finally, you will migrate your new contract by running `truffle migrate --network matic --reset`.

App and functionality development

Once you've deployed your new smart contract, let's get started on building out the application.

1. We'll be using the `react-hook-form` library to build a simple form experience. You'll need to install it by running `npm install react-hook-form`.
2. Next, create a new directory in `client/src` titled `components`.
3. Within `components`, create a file called `AllowlistForm.js` and paste in the following:

```
import React, { useEffect } from "react";
import { useForm } from "react-hook-form";

export default function AllowlistForm(props) {
  const {
    register,
    handleSubmit,
```

```
    setValue,
    formState: { errors },
  } = useForm();

  useEffect(() => {
    register("firstName", { required: true });
    register("lastName", { required: true });
    register("walletAddress", { required: true });
  }, [register]);

  async function submitForm(data) {
    console.log(data);
  }

  return (
    <div className="wrapper">
      <form onSubmit={handleSubmit((data) => submitForm(data))}>
        <div className="header">
          <h1>Allowlist Form</h1>
          <p>
            Please fill out this form to get allowlisted for this exclusive
            project.
          </p>
        </div>
        <label htmlFor="firstName">First Name</label>
        <input
          id="firstName"
          onChange={(e) => setValue("firstName", e.target.value)}
        />
        {errors.firstName && (
          <span role="alert" className="errorField">
            First name is required.
          </span>
        )}
        <label htmlFor="lastName">Last Name</label>
        <input
          id="lastName"
          onChange={(e) => setValue("lastName", e.target.value)}
        />
        {errors.lastName && (
          <span role="alert" className="errorField">
            Last name is required.
          </span>
        )}
        <label htmlFor="walletAddress">Wallet Address</label>
        <input
          id="walletAddress"
          onChange={(e) => setValue("walletAddress", e.target.value)}
        />
        {errors.walletAddress && (
          <span role="alert" className="errorField">
            Wallet address is required.
          </span>
        )}
        <input type="submit" className="submitButton" />
      </form>
    </div>
  );
}
```

The above code creates a component called `AllowlistForm` using the `react-hook-form` library installed earlier. The form allows a `firstName`, `lastName`, and `walletAddress` field, and also explicitly specifies what fields are required or not for error validation. The component also contains some error-handling logic for leaving required inputs blank and attempting to submit.

4. Next, to make the form look better, create a file called `AllowlistForm.css` and add in the following CSS code:

```
h1 {
  border-bottom: 1px solid white;
  color: #3d3d3d;
  font-family: sans-serif;
  font-size: 20px;
  font-weight: 600;
  line-height: 24px;
  text-align: center;
}
```

```
.header {
  margin-bottom: 10px;
}

form {
  background: white;
  border: 1px solid #dedede;
  display: flex;
  flex-direction: column;
  justify-content: space-around;
  margin: 0 auto;
  margin-top: 10px;
  max-width: 500px;
  padding: 30px 50px 0px;
}

input {
  border: 1px solid #d9d9d9;
  border-radius: 4px;
  box-sizing: border-box;
  padding: 10px;
  width: 100%;
  margin-bottom: 10px;
}

label {
  color: #3d3d3d;
  display: block;
  font-family: sans-serif;
  font-size: 14px;
  font-weight: bold;
  margin-bottom: 5px;
  text-align: left;
}

.errorField {
  color: red;
  font-family: sans-serif;
  font-size: 12px;
  margin-bottom: 10px;
  text-align: left;
}

.submitButton {
  background-color: #6976d9;
  color: white;
  font-family: sans-serif;
  font-size: 14px;
  margin: 20px 0px;
}
```

:::tip

To implement the form field, button, and input styling and color scheme, as well as to help outline form functionality, you can use [Retool tutorial]() on `react-hook-form` as a reference.

:::

:::note

If you receive a warning about old stylesheets, this is probably due to a legacy version of `create-react-app` that Truffle is using to set up the React project. To mitigate this, you can update your `package.json` 's `browserlist` property so it looks like this:

```
"browserslist": {
    "production": [
      ">0.3%",
      "not ie 11",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version",
      ">0.3%",
      "not ie 11",
      "not dead",
      "not op_mini all"
```

```
    ]
  }
```

:::

5. To wire up the styling with the component, add the following import statement at the top of your `AllowlistForm.js` component:

```
import "./AllowlistForm.css";
```

6. Finally, we need to actually surface the form in our demo app. We'll do this by updating the `App.js` file in the `client` directory. Replace the current contents of `App.js` with the following:

```
import React, { Component } from "react";
import AllowlistContract from "./contracts/Allowlist.json";
import getWeb3 from "./getWeb3";
import AllowlistForm from "./components/AllowlistForm";

import "./App.css";

class App extends Component {
  state = { storageValue: 0, web3: null, accounts: null, contract: null };

  componentDidMount = async () => {
    try {
      // Get network provider and web3 instance.
      const web3 = await getWeb3();

      // Use web3 to get the user's accounts.
      const accounts = await web3.eth.getAccounts();

      // Get the contract instance.
      const networkId = await web3.eth.net.getId();
      const deployedNetwork = AllowlistContract.networks[networkId];
      const instance = new web3.eth.Contract(
        AllowlistContract.abi,
        deployedNetwork && deployedNetwork.address
      );

      // Set web3, accounts, and contract to the state, and then proceed with an
      // example of interacting with the contract's methods.
      this.setState({ web3, accounts, contract: instance });
    } catch (error) {
      // Catch any errors for any of the above operations.
      alert(
        `Failed to load web3, accounts, or contract. Check console for details.`
      );
      console.error(error);
    }
  };

  render() {
    if (!this.state.web3) {
      return <div>Loading Web3, accounts, and contract...</div>;
    }
    return (
      <div className="App">
        <AllowlistForm
          contract={this.state.contract}
          accounts={this.state.accounts}
        />
      </div>
    );
  }
}

export default App;
```

You might notice that some of this `App.js` logic is similar to what was originally set up by Truffle! Our new-and-improved `App.js` will do the following:

- On page load, initialize a web3 instance, any related accounts, and an instance of our `Allowlist` smart contract (pulled from `Allowlist.json`, an instance of our contract that is generated from the previously run `truffle migrate` command). Within these initializations, we set corresponding state variables for the contract, account list, and web3 instances.
- When the page renders, if we're still fetching all of the web3 information, it'll show a `Loading message` rather than the form - this is determined by checking whether the web3 state variable has been set. Once that web3 state variable has been set, we return the `AllowlistForm` component and pass in our `contract` and `accounts` state variables - we'll need them within our component to write to the blockchain when we submit the form.
- To get a beautiful full-page lavender background like in the demo GIF, add the following at the bottom of your `App.css` file:

```
html,
body {
  width: 100%;
  height: 100%;
  background: lavender;
}
```

When navigating to http://localhost:3000 now, your application will look like this:

img

At this point, you can test the functionality of your form - try to submit empty responses to see if the error messages pop up. You can also fill out each field with random strings to test successfully submitting - to determine if it was successful, check your console, and you should see your submitted form responses as they come through.

Fauna setup and wiring

Now that the application is set up, hop into Fauna and connect it to our application.

1. Log into your Fauna account. You'll be taken to the main dashboard, where you'll create a database by clicking the "Create Database" button. Create a database with a name of your choice and make sure to select the "Classic" Region Group:

img

Once you create your database, you will be taken to the overview page.

2. Within your newly created database, you're going to create a Collection. Collections in Fauna are equivalent to the concept of a "table" in a traditional database - a dedicated place to store entries that all have the same information and fields as one another. You'll click the "New Collection" button and create a Collection with a name of your choosing:

img

:::tip

Something super interesting about Fauna is that you don't actually create `columns`, which, in a traditional database, are a way to structure entries that you insert in the table. The data you insert into Fauna determines the collection structure rather than the other way around.

:::

3. Once a database and a collection within it has been made, we need to generate a secret key to be able to access it in-app. To do so, go to the **Security** option in your database and click **New Key**. Create a new key with the following settings and give it a name of your choosing:

img

Click **Save** and copy the Secret that appears above the list of keys. Then, create a `.env` file in your project within client - this is a React-specific environment file that the frontend will read from. Add a `REACT_APP_FAUNADB_SECRET` environment variable and set it equal to the Secret that you copied when generating the API key. Your `.env` should look like this:

```
REACT_APP_FAUNADB_SECRET=your secret from fauna goes here
```

:::note

For React `.env` files, all custom environment variables MUST start with `REACT_APP_` in order for React to recognize them.

:::

4. Now that we have a collection ready in Fauna and an API key generated from the database it lives in, we can start writing to Fauna.

- In your project directory, run `npm install --save faunadb` to install `faunadb` locally.
- Next, in `client/src`, create an `api` folder.
- Within the `api` folder, create a file called `fauna.js`, which is going to be how we talk within our application to the Fauna API. Your `fauna.js` file should look like this:

```
require("dotenv").config(); // Load .env file
export async function addDocument(uuid, firstName, lastName, walletAddress) {
  const faunadb = require("faunadb");
  const q = faunadb.query;
  const client = new faunadb.Client({
    secret: process.env.REACT_APP_FAUNADB_SECRET,
    domain: "db.fauna.com",
    scheme: "https",
  });
  var response = client.query(
    q.Create(q.Collection("allowlist_members"), {
      data: {
```

```
        uuid: uuid,
        f_name: firstName,
        l_name: lastName,
        wallet_address: walletAddress,
      },
    })
  );

  return response;
}
```

The `fauna.js` starts by pulling the environment variable set in the `client/.env`. Then, it declares an `addDocument` function that takes in a `uuid` (which we will generate programmatically in a section below), and the form-inputted `firstName`, `lastName`, and `walletAddress`. Within the function, it initializes an instance of a Fauna API client utilizing the secret stores in the `.env`, which specifically will point Fauna to the database you created.

Finally, the function actually queries the API client to add a new item, called a Document, to the collection you created (in this code example it is pulling from a collection called `allowlist_members` - you'll need to change this to the name you chose for your own collection).

Finally, the API response is returned once the query is executed.

:::tip

The `faunadb` [package website](#) is a great resource for insight into configuring and connecting Fauna to the app!

:::

Submit form responses to Fauna

With the Fauna API client added in and a way to add new items to our collection, let's wire it up to our form!

If you recall from the user flow, we want the following things to happen when someone submits the form:

1. All the information is sent to the blockchain, but the private values are replaced with generated UUIDs.
2. Those private values (first name, last name, and wallet address) are instead sent to Fauna and stored next to that UUID.

Now that we have the functionality in place to send information to Fauna, let's use it to accomplish goal #2.

1. Add the following import at the top of your `AllowlistForm.js` file:

```
`import { *addDocument* } from "../api/fauna";`
```

2. Then, inside the `submitForm` function in `AllowlistForm.js`, replace the `console.log` statement with the following:

```
// add data to Fauna
const uuid = crypto.randomUUID();
await addDocument(uuid, data.firstName, data.lastName, data.walletAddress)
  .then((res) => {
    console.log(res);
  }
)
```

In this snippet, we are programmatically generating a UUID using the built-in `crypto.randomUUID()` [function](#), which will generate a 36-character-long v4 UUID. Once the UUID has been generated, we call the `addDocument` function we just added. Within `addDocument`, we're passing information from the form submission (captured in `data`, which is being initially passed into `submitForm` - `data.firstName`, `data.lastName`, and `data.walletAddress`) as well as the generated UUID. Finally, the API response is logged to the console, which will help us determine if the API call successfully went through.

3. At this point, your form is now configured to send information to Fauna when submitting a fully filled out form. You can test this by filling out each field in the form and submitting it. It should log an object in the console with a `data` property inside it that shows all the information submitted to Fauna. To confirm that it successfully submitted, you can pull up your collection in Fauna and see if the entry is there. It should look something like this:

img

Handling UUID collisions

Our application is now sending responses to Fauna. However, since we're using a randomly generated UUID, there is a chance that we could have duplicated UUID entries in our database. We want to make sure that before we store any information in Fauna that there are no other UUIDs that match the one we've generated, and if there are, we'll need to generate a new one.

1. In order to search the data we've been populating into our Fauna collection, we can use an [Index](#). Indexes in Fauna allow you to browse and easily search the data stored in your collections. To do this, you can go to your collection in Fauna and click the "New Index" button.

2. You will be taken to a page to create a new index - make sure the Source Collection field has correctly populated the collection you're trying to add to. Indexes are used for querying, so try to name the index something that specifically captures what it's going to do - since the index will be used to query entries by UUID, we chose `allowlist_members_by_uuid`. To specify what properties you'll be querying by, add `data.uuid` into the Terms field - this lets Fauna know to pull items based on the UUID. Lastly, since you do want the UUID to be unique, check the "Unique" option and also make sure the "Serialized" option also stays checked:

img

3. Once the index has been created, we can now query it from Fauna. To do this, we'll need to hop into our client/src/api/fauna.js file and add a new function, findUUID, to search by index when given a UUID passed in as a parameter:

```
export async function findUUID(uuid) {
  const faunadb = require("faunadb");
  const q = faunadb.query;
  const client = new faunadb.Client({
    secret: process.env.REACT_APP_FAUNADB_SECRET,
    domain: "db.fauna.com",
    scheme: "https",
  });
  client
    .query(q.Paginate(q.Match(q.Index("allowlist_members_by_uuid"), uuid)))
    .then((res) => {
      if (res.data.length === 0) {
        return false;
      } else {
        return true;
      }
    });
}
```

Very similarly to the `addDocuments` function, we need to do some initial setup and configurations with the Fauna client in this function. Then, we're calling a query that looks into a specific index (in this case `allowlist_members_by_uuid`) and sees if there are any matching items that have the `uuid` passed in as a function argument. In the event that there are no matches, an empty array is returned, so the function then checks for an empty array and returns `false` (indicating the UUID has not been found and is therefore not taken) - otherwise, it'll return `true` (indicating the UUID has a match).

4. Now that we have a way to check for existing UUIDs, we can add this in to our `submitForm` function in our `AllowlistForm.js` component.
   - To do this, we'll add the `findUUID` function in to the `../api/fauna` import statement at the top of the file like so:

     ```
     import { addDocument, findUUID } from "../api/fauna";
     ```

   - Replace the const uuid = crypto.randomUUID(); line with the following logic:

     ```
     // generate uuid
     let uuid = "";
     // check for duplicate uuids in db
     while (true) {
       const generatedUUID = crypto.randomUUID();
       const didFindUUID = await findUUID(generatedUUID);
       if (!didFindUUID) {
         uuid = generatedUUID;
         break;
       }
     }
     ```

   With the above code, we're initializing a UUID variable. Then, we're diving into an infinite loop that will run until it is manually broken out of - within the loop, a UUID is generated using `crypto.randomUUID()` . The generated UUID is then passed into a call to the `findUUID` function, which will return either true or false to the variable `didFindUUID` once the function call is complete. If `didFindUUID` returns false, meaning there are no matches in the database for this generated UUID, then the UUID variable is set to the generated UUID and the loop is broken out of. Otherwise, the entire process is repeated until a non-duplicate UUID is identified.

Submit form responses to the blockchain

The final step in our application is to write to the blockchain using our smart contract. As we mentioned above, we want the generated UUID that we store in Fauna to replace all of the actual private data (in this case, the first name, last name, and wallet address submitted). This adds a layer of security to the private data - it's not publicly available on the blockchain, only the UUID is, and you'd need access to the Fauna database to be able to see the information behind that UUID.

1. Since we want the form to submit to the blockchain only if the Fauna API call is successful (and only once it happens), we can nest the API call to add a new allowlister within the Fauna API call where we add a new document. We'll do this in the `submitForm` function by replacing the `console.log(res)` line so that it calls our `_createAllowlister` method from our smart contract:

```
async function submitForm(data) {
  // generate uuid
    let uuid = "";
  // check for duplicate uuids in db
  while (true) {
    const generatedUUID = crypto.randomUUID();
    const didFindUUID = await findUUID(generatedUUID);
    if (!didFindUUID) {
```

```
      uuid = generatedUUID;
      break;
    }
  }
  // add data to Fauna
  await addDocument(uuid, data.firstName, data.lastName, data.walletAddress)
    .then((res) => {
      // add data to contract
      props.contract.methods
        ._createAllowlister(uuid)
        .send({ from: props.accounts[0] })
    }
  )
}
```

Since we passed the `Allowlist` smart contract and all available accounts into our `AllowlistForm` component, we're able to access them to pull information from our smart contract. Since they're passed in as props, we're referencing them as `props.contract` and `props.accounts` rather than `contract` and `accounts`.

2. Once that has been added in, you can see if your application is successfully submitting to Fauna and the blockchain by filling out the form in your application. A MetaMask window should pop up, and if you click "Accept", you'll receive a notification after a few seconds that the transaction was successfully submitted. You can also check for the newly added document in your Fauna collection to make sure it's writing to both Fauna and the blockchain.

Read transaction data from the blockchain and lookup in Fauna

Now that we're able to write transactions to Fauna and then the blockchain, how can we actually find the data we need from a transaction we've submitted?

Luckily, the public nature of the blockchain makes it easy to track down transactions and also find the metadata associated with a given transaction.

1. We'll be using the Mumbai Polygonscan tool to find all the transactions associated with the wallet we've used to develop the application. This tool allows you to look up any wallet on the Mumbai-Testnet by its address and see a history of all of its transactions as well as the amount of MATIC inside the wallet.
   - To find your wallet, you can go to this url: https://mumbai.polygonscan.com/address/[put your wallet address here]
2. Once you navigate to this page, you'll see a list of all of the transactions associated with your wallet. The transactions display from most to least recent.

Click on a transaction to see more information by clicking on the transaction hash - this hash is uniquely generated and is what identifies your transaction.

img

3. When you click on the transaction hash, you'll go to a page that lists the transaction's specific details. To grab the information you'll need, click the "Click to see more" option near the bottom of the displayed transaction information.
4. Within the "Click to see more" option, you'll see a lot more information. We're going to scroll to the "Input Data" section, where there will be a long hash.

img

We can write a script to parse the transaction hash for the Input Data, so that we can turn that hash into actual data that we can pull our `uuid` from to be able to query in Fauna.

5. To decode this, we can use the abi-decoder library. In the `client` directory of your project, run the following to install the library:

```
npm install abi-decoder
```

6. Then, create a folder named `scripts` in the `client` directory.

7. Within the `scripts` folder, create a file called `decode-transaction.js` and paste the following information in:

```
const abiDecoder = require('abi-decoder');

// pulled from Allowlist.json "abi" field
const testABI = [
    {
      "constant": false,
      "inputs": [
        {
          "internalType": "string",
          "name": "_uuid",
          "type": "string"
        }
      ],
      "name": "_createAllowlister",
      "outputs": [],
      "payable": false,
      "stateMutability": "nonpayable",
      "type": "function"
    }
];
```

```
abiDecoder.addABI(testABI);

// add in your input data hash as a string here
const testData = "";
const decodedData = abiDecoder.decodeMethod(testData);
console.log(decodedData);
```

The above code will create an instance of the abi-decoder library so it can be used, storing it in the variable `abiDecoder`. In order to read the Input Data hash, the library requires an ABI, or Application Binary Interface, to tell the decoder what information, such as functions and arguments, is in the smart contract being used for the transaction. The ABI will help inform the decoder what information can be parsed from the hash.

If you used the same smart contract from this tutorial in your application, then the ABI has already been populated for you in the `testABI` variable. This ABI was taken from the smart contract metadata for `Allowlist.sol` that was generated upon migration. The metadata can be found in `client/contracts/Allowlist.json`. Within this file, there is an `abi` property, representing the `Allowlist` contract ABI, which is where the `testABI` value was pulled from.

The final touch for this program to run will be to add in the input data hash for the transaction you want to decode. Copy the Input Data you found from the Mumbai Polygonscan website and set it as a string equal to `testData`.

8. Once all information has been added in to the script, we can run it in the command line to see the output. Run the script from the `client` folder using this command:

```
node scripts/decode-transaction.js
```

You'll receive a response back that looks like this:

```
{
  name: '_createAllowlister',
  params: [
    {
      name: '_uuid',
      value: '2bb542ef-39b7-4991-bfa8-aac848fdce39',
      type: 'string'
    }
  ]
}
```

This response indicates that the transaction was initialized by the `_createAllowlister` function from the smart contract and that the `_uuid` value was passed in as a parameter (as well as what that value was).

Our `uuid` has been successfully identified from the transaction - in this particular response, it's `2bb542ef-39b7-4991-bfa8-aac848fdce39`.

9. Now that we have the `uuid`, we can search in Fauna using it to find the data behind the transaction. To do so, navigate to your Fauna Dashboard, click on the Database you used for your application, and then click on "Indexes" in the sidebar for that database. Make sure that the index you used for the app is displaying at the top of the page - if not, then select it from the list of indexes below to ensure you're querying from the right one.

Once you have the right index pulled up, paste your UUID into the field under "data.uuid" and make sure the dropdown above the field says "String".

When you click the search button, an entry should appear below the search field:

img

Once you expand the entry, you'll see all of the original data passed in to the form for that specific transaction, such as the first name, last name, and wallet address.

Error and success messaging

We've written a React form application that stores its data directly in Fauna and then submits it as a transaction to the blockchain where its private details are stored as a UUID, which can be used to reference the record in Fauna. We also showed how to take a transaction from the blockchain, parse its input data, and use it to search up the records in Fauna.

However, while working through form submission and testing everything in the application, you might've noticed it was sometimes difficult to tell when a record was successfully submitted or not. To mitigate this, we can add in some error and success messaging that will surface to indicate an errored form submission (either from a Fauna or Polygon issue) or a successful form submission. This part of the tutorial will focus on changing some API call logic in `AllowlistForm.js`.

Error Messaging

Ideally, we'd want to catch when the Fauna API throws an error, and we'd also want to catch when interacting with the blockchain throws an error - either of these scenarios indicates that the transaction won't go all the way through. We also want to make sure that if Fauna fails, it doesn't still try to interact with the blockchain - this could create inconsistency in the entries.

1. To do this, let's add a state variable to track whether or not the form submission has failed at any point. To initialize new state variables, we'll need to start by importing the `useState` hook, which can be added in where we're importing `React` and `useEffect`:

```
import React, { useEffect, useState } from "react";
```

2. Once imported, we can initialize our variable. We'll call it `formField` - this will be added in after the `useForm()` usage at the top of the component:

```
const {
    register,
    handleSubmit,
    setValue,
    formState: { errors },
} = useForm(); // this is already in the code, add the line below
const [formFail, setFormFail] = useState(false);
```

To initialize a new state variable, we need to declare a name (`formFail`) and also a function to set it (`setFormFail`). We also need to set it to a default value (since `formFail` will be a boolean indicating whether or not the form has failed, we can set the default value to `false`).

3. Now, we need to make sure that we add in error messaging that uses this state variable. If `formFail` becomes `true` , we'd want to surface the error messaging. We can add in this snippet of `jsx` within the `wrapper` div class in the `return()` function at the bottom of the file:

```
{formFail && (
  <div className="errorMessage">
    <p>Failed to submit allowlist entry. Please try again.</p>
  </div>
)}
```

4. In addition to adding the error message block, we need to add some styling for the error message in `AllowlistForm.css` :

```
  .errorMessage {
    background-color: #fe6f5e;
    border: solid 1px;
    border-color: #000000;
    margin-bottom: 10px;
    margin: 10px;
  }
```

5. Finally, we need to actually set `formFail` to `true` in areas where the form fails. To do this, we'll hop back into `AllowlistForm.js` and update our `submitForm` function's API calls to Fauna and the blockchain so that they have `.catch()` statements:

```
async function submitForm(data) {
    ... // some logic exists here
    // update this snippet that is already within submitForm()
    await addDocument(uuid, data.firstName, data.lastName, data.walletAddress)
      .then((res) => {
        if (!res.ok && res.status >= 400) {
          setFormFail(true);
          return;
        } else {
          // add data to contract
          props.contract.methods
            ._createAllowlister(uuid)
            .send({ from: props.accounts[0] })
            .catch(() => {
              setFormFail(true);
              return;
            });
        }
      })
      .catch(() => {
        setFormFail(true);
        return;
      });
    }
}
```

In addition to adding `.catch` statements to each API call (`addDocument` and `_createAllowlister`), the code above also adds a block that checks for a scenario where Fauna may not throw an error, but does return an error response (meaning that the API does not return a `res.ok` and that the `res.status` is greater than or equal to a `400` response code, which indicates an error).

Your error messaging will look like this:

img

Success Messaging

1. Let's say we want a success message that surfaces what we submitted in the form within the message - to do so, we'll need to store what we submitted in the form (first name, last name, and wallet address, which are all strings) as state variables, as well as one to track a successful form submission. To do so, add the following state variable initializations into your code:

```
const [firstName, setFirstName] = useState("");
const [lastName, setLastName] = useState("");
const [walletAddress, setWalletAddress] = useState("");
const [formSuccess, setFormSuccess] = useState(false);
```

2. Within the `wrapper` div in the `return()` in `AllowlistForm.js`, just as you added a `jsx` block for the error message, add one for the success message above or below it. The message will surface the form input-related state variables:

```
{formSuccess && (
  <div className="successMessage">
    <p>
      Successfully submitted allowlist entry for{" "}
      {firstName + " " + lastName} with wallet address {walletAddress}!
    </p>
  </div>
)}
```

3. Add some styling for the success message in `AllowlistForm.css`:

```
.successMessage {
  background-color: #e2fee2;
  border: solid 1px;
  border-color: #000000;
  margin-bottom: 10px;
  margin: 10px;
}
```

4. Finally, update all of your newly added state variables in a place where, after the API calls are executed, the form is considered fully submitted:

```
async function submitForm(data) {
    ... // some logic exists here
  await addDocument(uuid, data.firstName, data.lastName, data.walletAddress)
    .then((res) => {
      if (!res.ok && res.status >= 400) {
        setFormFail(true);
        return;
      } else {
        // add data to contract
        props.contract.methods
          ._createAllowlister(uuid)
          .send({ from: props.accounts[0] })
          .then(() => {
                        // update this snippet that is already within submitForm()
            setFirstName(data.firstName);
            setLastName(data.lastName);
            setWalletAddress(data.walletAddress);
            setFormSuccess(true);
          })
          .catch(() => {
            setFormFail(true);
            return;
          });
      }
    })
    .catch(() => {
      setFormFail(true);
      return;
    });
}
```

The form submission would be considered successful once both the Fauna and Polygon API calls went through. Because of this, we've added the logic to set the `firstName`, `lastName`, `walletAddress`, and `formSuccess` variables as a `.then()` block attached to `_createAllowlister` - this is the last API call that happens before the `submitForm` function ends, and adding in a `.then()` ensures that the states are only updated once the last API call has been completed.

Your success messaging will look like this:

img

Conclusion

We've built an allowlisting application in React that takes in private information, securely stores it in a Fauna database, and then submits a public UUID for each property corresponding to the transaction's database record to the Polygon blockchain. The full GitHub repository for this tutorial can be found here.

If you're looking to build off of this tutorial or expand the app from here, some great next steps would be to:

- Write some tests for the Solidity smart contracts
- Write some tests for the React functionality and components
- Try adding some additional features, such as:
  - Surfacing every submitted allowlister and the corresponding UUID info as an easily referenceable guide to find them in Fauna
  - Adding in logic so that if the Polygon API call fails for any reason, then it'll remove the entry from the Fauna database
  - Adding in logic so that if someone gets a failed submission, and then resubmits and gets a successful submission, the error message goes away and only shows the success message and vice versa

---

id: did-implementation title: Polygon DID Implementation sidebar_label: Identity description: Learn about DID implementation on Polygon keywords: - docs - polygon - matic - DID - identity image: https://wiki.polygon.technology/img/polygon-logo.png slug: did-implementation/getting-started

This is a startup guide for users who wish to use the implementation packages published by Polygon team, to generate and publish a Polygon DID on the Polygon ledger.

The Polygon DID method Implementation comprises of 3 packages, namely the polygon-did-registrar, polygon-did-resolver and polygon-did-registry-contract. A user who wants to incorporate the functionality to either register or read a DID on or from Polygon network can use the following guide.

A DID is essentially a unique identifier, that has been created without the presence of a central authority. DID in context of Verifiable Credentials is used to sign documents, thereby facilitating the user to prove ownership of the document when required.

Polygon DID Method

The Polygon DID method definition conforms to the DID-Core specifications and standards. A DID URI is composed of three components separated by colons, the scheme, followed by the method name and finally a method specific identifier. For Polygon the URI looks like:

```
did:polygon:<Ethereum address>
```

Here the scheme is `did`, method name is `polygon` and method specific identifier is an ethereum address.

Polygon DID Implementation

Polygon DID can be implemented with help of two packages, user can import the respective npm libraries and use them to incorporate Polygon DID methodologies in their respective applications. Details for implementation are provided in next section.

To get started, one first needs to create a DID. Creation in case of Polygon did is an encapsulation of two steps, first where a user needs to generate a DID uri for themselves and next register it on Polygon ledger.

Create DID

In your project to create a polygon DID URI one first needs to install:

```
npm i @ayanworks/polygon-did-registrar --save
```

Once the installation is completed, the user can use it as follows:

```
import { createDID } from "@ayanworks/polygon-did-registrar";
```

The `createdDID` function helps user generate a DID URI. While creating a DID, there can be two scenarios.

1. The user already owns a wallet and wishes to generate a DID corresponding to the same wallet.

```
```
const {address, publicKey58, privateKey, DID} = await createDID(network, privateKey);
```
```

2. If the user does not have an existing wallet and wants to generate one, the user can use:

```
```
const {address, publicKey58, privateKey, DID} = await createDID(network);
```
```

The network parameter in both cases refers to whether the user wants to create the DID on Polygon Mumbai Testnet or Polygon Mainnet.

Sample Input:

```
network :"testnet | mainnet"
privateKey? : "0x....."
```

After creating DID, you will have a DID URI generated.

```
DID mainnet: did:polygon:0x...
DID testnet: did:polygon:testnet:0x...
```

Register DID

To register the DID URI and the corresponding DID document on ledger, the user first needs to use `polygon-did-registrar` as follows:

```
import { registerDID } from "@ayanworks/polygon-did-registrar";
```

As a prerequisite to registering DID, the user needs to make sure that the wallet corresponding to the DID has the necessary tokens balance available. Once the user has a token balance in the wallet, a call can be made to the registerDID functionality as shown below:

```
const txHash = await registerDID(did, privateKey, url?, contractAddress?);
```

Parameters `did` and `privateKey` are mandatory, while it is optional to enter the `url` and the `contractAddress`. If the user does not give the last two parameters, the library picks up the default configurations of the network from the DID URI.

If all the parameters match the specifications and everything is given in correct order, the `registerDID` function returns a transaction hash, a corresponding error is returned otherwise.

And with this, you have successfully completed your task of registering a DID on the Polygon Network.

Resolve DID

To start, install the following libraries:

```
npm i @ayanworks/polygon-did-resolver --save
npm i did-resolver --save
```

To read a DID document registered on ledger, any user with a DID polygon URI can first in their project import,

```
import * as didResolvers from "did-resolver";
import * as didPolygon from '@ayanworks/polygon-did-resolver';
```

After importing the packages, the DID document can be retrieved by using:

```
const myResolver = didPolygon.getResolver()
const resolver = new DIDResolver(myResolver)

const didResolutionResult = this.resolver.resolve(did)
```

where the `didResolutionResult` object is as follows:

```
didResolutionResult:
{
    didDocument,
    didDocumentMetadata,
    didResolutionMetadata
}
```

It should be noted that, no gas cost will be entailed by the user while trying to resolve a DID.

Update DID Document

To encapsulate the project with the ability to update the DID document, the user first needs to use `polygon-did-registrar` as follows:

```
import { updateDidDoc } from "@ayanworks/polygon-did-registrar";
```

Next, call the function:

```
const txHash = await updateDidDoc(did, didDoc, privateKey, url?, contractAddress?);
```

It should be noted that to update the DID document, only the owner of DID can send the request. The private key here should also hold some corresponding MATIC tokens.

If the user does not provide the configuration with `url` and `contractAddress`, the library picks up the default configurations of the network from the DID URI.

Delete DID Document

With Polygon DID implementation a user can also revoke his DID Document from the ledger. The user first needs to use `polygon-did-registrar` as follows:

```
import { deleteDidDoc } from "@ayanworks/polygon-did-registrar";
```

Then use,

```
const txHash = await deleteDidDoc(did, privateKey, url?, contractAddress?);
```

Amongst the parameters it is notable that, `url` and `contractAddress` are optional parameters, which if not provided by the user, a default configuration will be picked up by the function based on the DID URI.

It is important for the private key to hold the necessary Matic tokens, as per the network configuration of DID, or the transaction would fail.

Contributing to the Repository

Use the standard fork, branch, and pull request workflow to propose changes to the repositories. Please make branch names informative by including the issue or bug number for example.

```
https://github.com/ayanworks/polygon-did-registrar
https://github.com/ayanworks/polygon-did-resolver
```---
id: nft-tutorial
title: Polygon NFT Tutorial
sidebar_label: ERC721
description: Build, mint, and send around your own NFT (ERC721).
keywords:
  - graph
  - matic
  - docs
  - polygon
  - nft tutorial
  - mint nft on polygon
  - erc721
image: https://wiki.polygon.technology/img/polygon-logo.png
---

import useBaseUrl from '@docusaurus/useBaseUrl';

Build, mint, and send around your own ERC721 (NFT) on Mumbai (testnet) and then on Polygon mainnet!

Guides

1. Click [here](https://github.com/scaffold-eth/scaffold-eth/tree/matic) for the detailed tutorial in scaffold-eth that demonstrates [how to mint an NFT on Mumbai and then Polygon](https://github.com/primeshprimesh/firstSimpleNFTProject).

2. Click [here](https://docs.scaffoldeth.io/scaffold-eth/examples-branches/common-web3-patterns/push-the-button#side-quests) for the Multi-player Turn Based Game (Push The Button).

3. Click [here](https://docs.alchemy.com/alchemy/tutorials/nft-minter) for a tutorial on how to create an NFT Minter that enables users to mint NFTs from a front-end (a mini-version of OpenSea). Be sure to use your Polygon RPC URL from [Alchemy](https://alchemy.com/?a=polygon-docs) instead of the Ethereum version!

4. Click [here](https://docs.alchemy.com/alchemy/enhanced-apis/nft-api) for the Alchemy NFT API that will allow you to instantly find NFTs by owner, as well as verify and display NFTs within your app.

5. Click [here](https://nftschool.dev/tutorial/mint-nftstorage-polygon/) to mint with nft.storage [link](https://nft.storage/) and polygon.
---
id: basics-web3-development
title: Learn the Basics of Web3 Development
sidebar_label: Web3 Development Basics
description: Learn the Basics of Web3 Development.
keywords:
  - docs
  - matic
  - polygon
  - web3
  - development
  - basics
image: https://wiki.polygon.technology/img/polygon-logo.png

Learn Web3 Programs

- [Alchemy University (free)](https://university.alchemy.com)
- [ConsenSys Academy Developer Program On-Demand course](https://consensys.net/academy/ondemand/)
- [Dapp University](https://www.youtube.com/channel/UCY0xL8V6NzzFcwzHCgB8orQ)

Smart Contract & Token Tutorials

- [Deploy a Smart Contract on Polygon using thirdweb](https://blog.thirdweb.com/guides/deploy-a-smart-contract-on-polygon-matic/)
- [Hello World, smart contract tutorial](https://docs.alchemy.com/alchemy/tutorials/hello-world-smart-contract)
- [Deploy a smart contract using REMIX IDE](https://www.quicknode.com/guides/solidity/how-to-deploy-a-smart-contract-on-matic-polygon)
- [How to create a token (ERC20)](https://www.quicknode.com/guides/solidity/how-to-create-and-deploy-an-erc20-token)
- [How to create an NFT](https://docs.alchemy.com/alchemy/tutorials/how-to-create-an-nft)

```
## General Tutorials

- [Full stack dApp tutorial series](https://kauri.io/#collections/Full%20Stack%20dApp%20Tutorial%20Series/full-stack-dapp-tutorial-series-intro/)
- [Truffle tutorial](https://www.trufflesuite.com/tutorial)

## Basics of Ethereum Development

- [Web3 developer stack](https://www.quicknode.com/guides/web3-sdks/the-web3-developer-stack)
- [Integrating IPFS with Ethereum](https://www.quicknode.com/guides/web3-sdks/how-to-integrate-ipfs-with-ethereum)
- [What is Ethereum?](https://blockgeeks.com/guides/ethereum/)
- [Mastering Ethereum](https://github.com/ethereumbook/ethereumbook)
- [OpenZeppelin Learn Docs](https://docs.openzeppelin.com/learn/)
- [Getting started with Ethereum development](https://medium.com/alchemy-api/getting-started-with-ethereum-development-using-alchemy-c3d6a45c567f)
---
id: pos-faqs
title: Frequently Asked Questions
sidebar_label: Frequently Asked Questions
description: Common questions on validator operations
keywords:
  - docs
  - matic
  - polygon
  - validator
  - faq
image: https://wiki.polygon.technology/img/polygon-logo.png
---

import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';

<Tabs
defaultValue="node"
values={[
{ label: 'Node', value: 'node', },
{ label: 'Validator', value: 'validator', },
]
}>

<!-- ================================================================================================================ -->
<!-- ================================================================ NODE FAQs ================================================================ -->
<!-- ================================================================================================================ -->

<TabItem value="node">

:::tip

Keep up with the latest node and validator updates from the Polygon team and the community by subscribing to [<ins>Polygon notifications</ins>](https://polygon.technology/notifications/).

:::

### 1. Are the private keys the same for Heimdall and Bor keystore?

Yes, the private key used for generating Validator keys and Bor Keystore is the same. The private key used in this instance is your Wallet's ETH address where your Polygon testnet tokens are stored.

### 2. What are some Common Commands?

We currently have an easy to dive-in list for you for the Linux packages. We will keep updating this list regularly for more convenience.

<details>
<summary>For Linux packages</summary>

#### Configuration

- Where to find heimdall genesis file: `$CONFIGPATH/heimdall/config/genesis.json`

- Where to find heimdall-config.toml: `/etc/heimdall/config/heimdall-config.toml`

- Where to find config.toml: `/etc/heimdall/config/config.toml`

- Where to find heimdall-seeds.txt: `$CONFIGPATH/heimdall/heimdall-seeds.txt`

- How to start Heimdall: `$ sudo service heimdalld start`
```

```
- How to start Heimdall rest-server: `$ sudo service heimdalld-rest-server start`

- How to start Heimdall bridge-server: `$ sudo service heimdalld-bridge start`

- Where to find Bor genesis file: `$CONFIGPATH/bor/genesis.json`

- How to start Bor: `sudo service bor start`

#### Logs

- How to retreive Heimdall logs: `/var/log/matic-logs/`

- How to check heimdall logs: `tail -f heimdalld.log`

- How to check Heimdall rest-server: `tail -f heimdalld-rest-server.log`

- How to check Heimdall bridge logs: `tail -f heimdalld-bridge.log`

- How to check bor logs: `tail -f bor.log`

#### How to kill Bor process

1. **For Linux:**

    - `ps -aux | grep bor`. Get the PID for Bor and then run the following command.
    - `sudo kill -9 PID`

2. **For Binaries**:

Go to `CS-2003/bor` and then run, `bash stop.sh`

#### To get the latest peer details

```bash
bor attach bor.ipc

# Run within Bor terminal:
admin.peers.forEach(function(value){
    console.log(value.enode+',')
})

# Type 'exit' and enter to exit Bor terminal
```
```
### 3. Error: Failed to unlock account (0x...) No key for given address or file

This error occurs because the path for the `password.txt` file is incorrect. You can follow the below steps to rectify this:

This error occurs because the path for the `password.txt` and Keystore file is incorrect. You can follow the below steps to rectify this:

1. Copy the Bor keystore file to `/etc/bor/dataDir/keystore`
2. And `password.txt` to `/etc/bor/dataDir/`
3. Make sure you have added correct address in `/etc/bor/metadata`

For Binaries:

1. Copy the Bor keystore file to: `/var/lib/bor/keystore/`
2. And `password.txt` to `/var/lib/bor/password.txt`

### 4. Error: Wrong Block.Header.AppHash. Expected xxxx

This error usually occurs when the Heimdall service is stuck on a block; there is no reversal method available on Heimdall.

To resolve this, you need to reset Heimdall completely:

    sudo service heimdalld stop
    heimdalld unsafe-reset-all

After that, you should sync from the snapshot again:

    wget -c <Snapshot URL>
    tar -xzvf <snapshot file> -C <HEIMDALL_DATA_DIRECTORY>

Then, start the Heimdall services again.
```
```

5. From where do I create the API key?

You can access this link: https://infura.io/register . Make sure that once you have setup your account and project, you copy the API key for Ropsten and not Mainnet.

Mainnet is selected by default.

6. Heimdall isn't working. I'm getting a Panic error

Actual Error: My heimdalld isn't working. In the log the first line is: panic: Unknown db_backend leveldb, expected either goleveldb or memdb or fsdb

Change the config to `goleveldb` in config.toml

7. How do I delete remnants of Heimdall and Bor?

If you want to delete remnants of Heimdall and Bor then you can run the following commands Bor:

For Linux package: `$ sudo dpkg -i matic-bor` and delete Bor directory: `$ sudo rm -rf /etc/bor`

For Binaries: `$ sudo rm -rf /etc/bor` and `$ sudo rm /etc/heimdall`

8. How many validators can be active concurrently?

There will be upto 100 active validators at a time. We will bring in more participants if the limit is reached mid-way through the event as well. Note that active validators is mostly those whose uptime is high. Participants with high downtime will be forced out.

9. How much should I stake?

`stake-amount` and `heimdall-fee-amount` - how much it should be?

A minimum of 10 Matic tokens is required for the stake amount whereas heimdall fee should be greater than 10. For example, your stake amount is 400 then the heimdall fee should be 20. We suggest to keep the Heimdall fee as 20.

However, please note that the values entered in `stake-amount` and `heimdal-fee-amount` should be entered in 18 decimals. For example:

```
heimdallcli stake --staked-amount 40000000000000000000  --fee-amount 1000000000000000000 --validator
0xf8d1127780b89f167cb4578935e89b8ea1de774f
```

10. I was selected to become a validator but my ETH address was incorrect. What do I do?

If you have access to the ETH address that you submitted earlier then you can transfer the Test tokens from that account to the current account. And then you can initiate your process of setting up your nodes.

If you don't have access to that ETH address, we won't be transferring you tokens separately. You can re-register in the form again with the correct ETH address.

11. I'm getting an error starting the bridge

Error: Object "start" is unknown, try "bridge help". Is it still ok to ignore this?

Check "which bridge" - if it's `/usr/sbin/bridge` you're not running the right "bridge" program.

Try `~/go/bin/bridge` instead `(or $GOBIN/bridge)`

12. I'm getting dpkg error

Error: "dpkg: error processing archive matic-heimdall_1.0.0_amd64.deb (--install): trying to overwrite '/heimdalld-rest-server.service', which is also in package matic-node 1.0.0"

This occurs mainly because of a previous installation of Matic on your machine. To resolve you can run: `sudo dpkg -r matic-node`

13. I'm not clear on which Private Key should I add when I generate validator key

The Private key to be used is your Wallet's ETH address where your Polygon testnet Tokens are stored. You can complete the setup with one public-private key pair tied to the address submitted on the form.

14. Is there a way to know if Heimdall is synced?

You can run the following command to check it: `$ curl [http://localhost:26657/status](http://localhost:26657/status)`

Check the value of `catching_up` . If it is false, then the node is all synced up.

15. What if someone become a Top 10 staker, how he will receive his MATIC reward at the end?

Stage 1 rewards are not based on stake. Participants with high stake don't automatically qualify for a reward in this stage.

16. What should be my heimdall version?

To check your Heimdall version, you can simply run: `heimdalld version`

The correct version of Heimdall for stage 1 should be `heimdalld version is beta-1.1-rc1-213-g2bfd1ac` .

17. What values should I add in the stake amount and fee amount?

A minimum of 10 Matic tokens is required for the stake amount whereas heimdall fee should be greater than 10. For example, your stake amount is 400 then the heimdall fee should be 20. We suggest to keep the Heimdall fee as 20.

However, please note that the values entered in `stake-amount` and `heimdal-fee-amount` should be entered in 18 decimals. For example:

```
heimdallcli stake --staked-amount 40000000000000000000  --fee-amount 1000000000000000000 --validator
0xf8d1127780b89f167cb4578935e89b8ea1de774f
```

18. What's the difference between `/var/lib/heimdall` and `/etc/heimdall?`

`/var/lib/heimdall` is the heimdall dir when you use the binary installation method. `/etc/heimdall` is for the Linux package installation method.

19. When I make the stake transaction, I'm getting "Gas Exceeded" error

This error may occur because of the stake or fee amount format. The values entered during the stake command need to have 18 decimals.

However, please note that the values entered in `stake-amount` and `heimdal-fee-amount` should be entered in 18 decimals. For example:

```
heimdallcli stake --staked-amount 40000000000000000000  --fee-amount 1000000000000000000 --validator
0xf8d1127780b89f167cb4578935e89b8ea1de774f
```

20. When will I get a chance to become a Validator?

We are progressively adding validators throughout the course of Stage 1 event. We will be releasing a list of new external validators gradually. This list will be announced on the Discord channel.

21. Where can I find Heimdall account info location?

For binaries: `/var/lib/heimdalld/config`

For Linux package: `/etc/heimdall/config`

22. Which file do I add the API key in?

Once you have created the API key, you need to add the API key in `heimdall-config.toml` file.

23. Which file do I add the persistent_peers?

You can add the persistent_peers in the following file: `/var/lib/heimdalld/config/config.toml`

24. Did you reset Tendermint without resetting your application's data?

In such a case, you can reset heimdall config data and try running the installation again.

```
$ heimdalld unsafe-reset-all
$ rm -rf $HEIMDALLDIR/bridge
```

25. Error: Unable to unmarshall config Error 1 error(s) decoding

Error: `* '' has invalid keys: clerk_polling_interval, matic_token, span_polling_interval, stake_manager_contract, stakinginfo_contract`

This occurs mostly because when there are typos, or some missing parts or an old config file which is still a remnant. You will need to clear all the remnants and then try setting it up again.

26. To stop Heimdall and Bor services

1. **For Linux packages**:
   - Stop Heimdall: `sudo service heimdalld stop`
   - Stop Bor: `sudo service bor stop` , or
     > *`ps -aux | grep bor` . Get the PID for Bor and then run the following command.*
     > *`sudo kill -9 PID`*
2. **For Binaries**:
   - Stop Heimdall: `pkill heimdalld`
   - Stop Bridge: `pkill heimdalld-bridge`
   - Stop Bor: Go to CS-2001/bor and then run, `bash stop.sh`

27. To remove Heimdall and Bor directories

1. **For Linux packages**:
   - Delete Heimdall: `sudo rm -rf /etc/heimdall/*`
   - Delete Bor: `sudo rm -rf /etc/bor/*`
2. **For Binaries**:

- Delete Heimdall: `sudo rm -rf /var/lib/heimdalld/`
- Delete Bor: `sudo rm -rf /var/lib/bor`

28. What to do when you get "Wrong Block.Header.AppHash." error

This error usually occurs due to Infura requests getting exhausted. When you setup a node on Polygon, you add an Infura Key to the Config file (Heimdall). By default you are allowed 100k Requests per day, if this limit is crossed, then you would face such problems. To resolve this you can create a new API key and add it to the `config.toml` file.

How can I reserve a Validator spot?

We currently manage validator evaluation through communications / interview before we onboard a validator into the Polygon network. We are also working on building an automated application and admission process, inclusive of questionnaire, scoring logic, and ranking to streamline the application process.

Once complete, this process of validator admission will be passed to the community for approval via governance, and subsequently implemented if agreed upon.

What are the different states a Validator can be in?

- **Active**: Validator is in the current validator set, produces blocks at the Bor layer, participates in Heimdall consensus and commits checkpoint transactions to the Ethereum mainnet.
- **Notice**: Validator sends a transaction to unbond. Before entering into the unbonding period, validator needs to be in active state creating, signing and proposing blocks for a certain time.
- **Unbonding**: Validator is inactive in this state and thus earns no reward.

Is there a minimum amount of MATIC required to stake to become a validator?

The minimum is 1 MATIC. We had earlier mentioned that we are thinking of having a minimum self stake requirement from the validators, as we do hope that validators also have their skin in the game.

However, since we will be moving to a robust replacement strategy as the number of validator slots are limited as of now, this does not need any minimum self stake requirement. It is however, logical that over time, the average / median stake by a validator will tend upwards and become substantial.

How can a new Validator replace an existing one?

There is limited space for accepting new validators. New validators can only join the active set when a currently active validator unbonds.

A new auction process for validator replacement will be rolled out.

Heimdall shows "Failed Sanity Checks"

`Addressbook` warnings can be ignored without an issue most of the time. If your node is connected to sufficient number of peers, these kind of errors can be ignored. Your `pex` is just trying to re-establish its connections with peers already present in `addrbook.json`.

Heimdall and Bor logs are fine and even my bridge is running correctly but my node is not signing any checkpoints

This could happen if you have missed adding the `ETH_RPC_URL` in the `heimdall-config.toml` file. Please check if you have added it. If not, ensure that you add the correct URL and then restart your Heimdall service.

Can I start Bor before Heimdall is completely synced?

No, you cannot. If you start your Bor without Heimdall being completely synced, you face issues on your Bor.

Validator Heimdall is unable to connect to peers

This typically means that your sentry Heimdall is running into issues. Check your sentry Heimdall and see if the service is running fine. If the service is stopped, then restarting the service on your sentry should resolve this issue. Similarly, after fixing your sentry, a restart of your Heimdall service should also resolve the problem.

Heimdall shows "pong timeout"

Full error:

```
E[2021-03-01|13:19:12.252] Connection failed @
sendRoutinemodule=p2ppeer=3d1f71344c2d3262eac724c22f8266d9b3e41925@3.217.49.94:26656 conn=MConn{3.217.49.94:26656} err="pong
timeout"
```

Usually restarting the Heimdall service should resolve the problem.

Heimdall shows "Error: Wrong Block.Header.AppHash. Expected xxxx"

This error usually occurs when Heimdall service is stuck on a block and there is no rewind option available on Heimdall.

Solution: To resolve this:

- Reset Heimdall completely

- Sync from the snapshot again

Reset Heimdall

Reset Heimdall with the following commands:

```
sudo service heimdalld stop
heimdalld unsafe-reset-all
```

Sync Heimdall from Snapshot

This is how you sync Heimdall from a Snapshot:

```
wget -c <Snapshot URL>
tar -xzvf <snapshot file> -C <HEIMDALL_DATA_DIRECTORY>
```

Then start Heimdall services again. Refer to: [Run a Validator Node]

Heimdall shows "dpkg: error processing archive"

Full error:

```
dpkg: error processing archive matic-heimdall_1.0.0_amd64.deb (--install): trying to overwrite '/heimdalld-rest-server.service',
which is also in package matic-node 1.0.0
```

This occurs mainly because of a previous installation of Polygon on your machine. To resolve you can run: `sudo dpkg -r matic-node`

It is not clear which private Key I should add when I generate a validator key

The private key to be used is your wallet's ETH address where your Polygon tokens are stored.

Is there a way to know if Heimdall is synced?

You can run the following command to check it:

```
curl http://localhost:26657/status
```

Check the value of `catching_up`. If it is `false`, then the node is all synced up.

Where can I find Heimdall account info location?

`/var/lib/heimdall/config`

Which file do I add the persistent_peers?

You can add the persistent_peers to `/var/lib/heimdall/config/config.toml`.

Heimdall shows “Did you reset Tendermint without resetting your application's data?”

Reset the Heimdall config data and try running the installation again:

```
heimdalld unsafe-reset-all
rm -rf $HEIMDALLDIR/bridge
```

Heimdall shows a Panic error

Full error:

```
panic: Unknown db_backend leveldb, expected either goleveldb or memdb or fsdb
```

Change the config to `goleveldb` in `config.toml`.

Are the private keys the same for Heimdall and Bor keystore?

Yes, the private key used for generating the validator keys and Bor keystore are the same. The private key used in this instance is your wallet's ETH address where your Polygon tokens are stored.

Error: (Heimdall) Please repair the WAL file before restarting module=consensus

This issue happens when the WAL file is corrupted.

Solution:

Run the following commands:

```
WALFILE=~/.heimdalld/data/cs.wal/wal
cp $WALFILE ${WALFILE}.bak
git clone https://github.com/maticnetwork/tendermint.git
cd tendermint
go run scripts/wal2json/main.go $WALFILE > wal.json
rm $WALFILE
go run scripts/json2wal/main.go wal.json $WALFILE
```

Bor shows 'Looking for peers' and cannot find peers

This could happen when Bor has lost connectivity with other peers. In the case of the validator, this occurs when the connectivity with the sentry has failed.

1. Open the file `~/var/lib/bor/config.toml` on your Sentry node.

2. Add the Trusted nodes, Static Nodes, and

```
[Node.P2P]

TrustedNodes = ["enode://<enode_id_of_validator_node>"]

[Node.P2P]
BootNodes= ["enode://static_node_enode1@ip:port", "enode://static_node_enode2@ip:port", ... ]

StaticNodes = ["enode://static_node_enode1@ip:port", "enode://static_node_enode2@ip:port", ... ]

TrustedNodes = ["enode://<enode_id_of_validator_node>"]
```

3. Now restart Bor: `sudo service bor restart`

Follow the same steps on your validator node and instead of `enode_id_of_validator_node` , you need to provide the `enode_id_of_sentry_node` in TrustedNodes.

:::note

Bootnodes & Static Nodes **can use the same values, are not required on the validator and apply only for sentry and full nodes.**

:::

Sentry Bor shows 'Looking for peers' and cannot find peers

This could happen when Bor has lost connectivity with other peers. Generally checking the `~/node/bor/start.sh` file should show you your bootnodes. Check if the bootnodes are entered correctly without any formatting issues. If you have made any changes to the file, then please restart your Bor service and check if the issue is resolved.

If the issue persists, contact support team on Discord.

Bor shows "Failed to prepare header mining at block 0"

This happens because of a formatting issue in your `/var/lib/bor/data/bor/static-nodes.json` file. Ensure there are no space and no additional characters like < / > . If you have made any changes to the file then please restart your Bor service and you should see logs printing.

Bor shows "30303 or invalid command: /home/ubuntu/.bor/password.txt"

This is because you have not created the Bor keystore and the password file for it. Ensure that you follow all the steps from the guide setup.

Bor shows "Impossible reorg, please file an issue"

Let these logs be. Your node should ideally not suffer because of this and the issue should be automatically resolved.

If your node is suffering because of this, please contact the support team on Discord.

Bor shows "Failed to prepare mining for header"

This message is not an error. The message indicates that the Bor node is not the one creating blocks right now.

Bor shows "Address is required as argument"

This means that you have not added your signer address to the metadata. You can add it using this path `/etc/matic/metadata` . Once the address is added, you can then restart the Bor service and everything should be fine.

Bor shows "Failed to unlock account (0x...) No key for given address or file"

This error occurs because the path for the `password.txt` file is incorrect.

To fix:

1. Copy the Bor keystore file to `/etc/bor/dataDir/keystore`
2. Copy the `password.txt` file to `/etc/bor/dataDir/`

3. Make sure you have added the correct address in `/etc/bor/metadata`

For binaries:

1. Copy the Bor keystore file to `/var/lib/bor/keystore/`
2. Copy `password.txt` file to `/var/lib/bor/password.txt`

Bor shows "unable to whitelist checkpoint err="missing checkpoint blocks""

This is not an issue. It denotes that the node is behind the last checkpoint and there can be many reasons for that (mostly if the node is actually syncing from scratch or snapshot).

Bor shows "Synchronisation failed, retrying"

It's a generic statement that synchronization can fail due to many reasons like bad peers, timeouts, etc.

Bor shows "Peer discarded announcement"

If a peer tries to serve the node with an old block, then the node doesn't accept it and it throws this message.

How to connect to a Web Socket?

Your /var/lib/bor/config.toml should have settings like this:

```
[jsonrpc]
ipcpath = "/var/lib/bor/bor.ipc"
# ipcdisable = false
# gascap = 50000000
# txfeecap = 5.0
[jsonrpc.http]
    enabled = true
    port = 8545
    host = "0.0.0.0"
    api = ["eth", "net", "web3", "txpool", "bor"]
    vhosts = ["*"]
    corsdomain = ["*"]
    # prefix = ""
[jsonrpc.ws]
    enabled = true
    port = 8546
    # prefix = ""
    host = "0.0.0.0"
    api = ["eth", "net", "web3", "txpool", "bor", "net"]
    origins = ["*"]
```

And then restart Bor for the changes to reflect.

Node is not signing any checkpoints

Your node not signing checkpoints could be for multiple reasons:

1. Check if your Heimdall service is running correctly on your sentry and validator nodes. If the service has stopped abruptly or you see any errors, try restarting your Heimdall service and see it comes back to normal. If the issue persists, contact support team on Discord.
2. Check your Bor service and see if it has halted abruptly or there are any errors on the logs. Try restarting your Bor service to resolve this issue. If the issue persists, contact support team on Discord.
3. Check if your Heimdall Bridge is running or not or if it has any errors in the logs. Try restarting the service and see if the issue resolves. If the issue persists, contact support team on Discord.

If none of this is the issue, contact support team on Discord.

How to set up a validator node on the mainnet?

See Getting Started.

How to set up a non-validating node?

Check out: Run a Validator Node

Why do I have to keep ETH in my signer account?

ETH is required on your signer account because for submitting checkpoints to Ethereum, all transactions require ETH to be used as gas. Hence ETH is required on your signer account.

Setting up a node with Ansible errors out with "Host not found"

This could be because your `inventory.yml` file may have some formatting issues. Correct them with proper indentation and then try again.

As a validator do I need to run both a sentry and a validator node?

Yes, you have to run both a sentry and a validator node.

The Polygon ecosystem and architecture demands that you run a sentry + validator setup to ensure that your validator node is not exposed to the public and only your sentry node is.

Your sentry node gleans information / blocks from the network and then relays them to the validator for validation.

What is the minimum disk space required to run a Validator node?

See Validator Node System Requirements.

Bridge shows "Error while fetching mainchain receipt error="

These are normal logs. Do not do anything to your bridge. Let it run as it is.

Validator Bor is stuck on a block for a long time

This means that your Bor on your sentry is also stuck because your validator gets information from your sentry.

Please check your Bor logs on your sentry and see if everything is okay.

Restart the Bor service one on your Bor and then simultaneously restart your Bor service on your validator as well.

Can I run multiple sentries for a validator?

Yes, you can.

Can I run multiple validators using the same signer key?

No. You cannot. Polygon's architecture currently does not allow validators running multiple validator nodes using the same signer key.

Is there a way to run a light Bor node?

There is no light node option as of now. Check out: Run a Full Node

What is the uptime percentage calculation on the staking dashboard?

It is calculated as per the last 200 checkpoints submitted to the ones you have actually signed.

What ports are to be kept open on the sentry node?

You will need to make sure that you open ports 22, 26656 and 30303 to world (0.0.0.0/0) on sentry node firewall.

What is the command to check the latest block height on Heimdall?

You can run this command `curl localhost:26657/status`.

What is the command to check the latest block height on Bor?

Run the following command:

```
curl  http://<your ip>:8545 -X POST -H "Content-Type: application/json" -d '
  {"jsonrpc":"2.0", "id":1, "method":"bor_getSigners", "params":["0x98b3ea"]}
'
```

Standard upgrade commands for Heimdall

```
cd ~/heimdall
git pull
git checkout <branch tag>
make install
sudo service heimdalld restart
```

The latest version, Heimdall v0.3.4, contains a few enhancements such as:

1. Restricting data size in state sync txs to:
   - **30Kb** when represented in **bytes**
   - **60Kb** when represented as **string**.
2. Increasing the **delay time** between the contract events of different validators to ensure that the mempool doesn't get filled very quickly in case of a burst of events which can hamper the progress of the chain.

The following example shows how the data size is restricted:

```
Data - "abcd1234"
Length in string format - 8
Hex Byte representation - [171 205 18 52]
Length in byte format - 4
```

Standard upgrade commands for Bor

These are the commands to upgrade Bor:

```
cd ~/bor
git pull
git checkout <branch tag>
sudo service bor stop
make bor-all
sudo service bor start
```

How do I delete remnants of Heimdall and Bor?

If you want to delete the remnants of Heimdall and Bor, you can run the following commands:

Bor:

For Linux package:

```
sudo dpkg -i matic-bor
```

And delete Bor directory:

```
sudo rm -rf /etc/bor
```

For Binaries:

```
sudo rm -rf /etc/bor
```

And:

```
sudo rm /etc/heimdall
```

Bridge shows "Object "start" is unknown"

Check `which bridge` — if it's `/usr/sbin/bridge`, you are not running the right "bridge" program.

Try `~/go/bin/bridge` instead (or `$GOBIN/bridge)`.

Error: Unable to unmarshall config Error 1 error(s) decoding

Full error:

```
'' has invalid keys: clerk_polling_interval, matic_token, span_polling_interval, stake_manager_contract, stakinginfo_contract
```

This occurs mostly because there are typos, some missing parts or an old config file which is still a remnant. You will need to clear all the remnants and then try setting it up again.

To stop Heimdall and Bor services

For Linux packages:

Stop Heimdall: `sudo service heimdalld stop`

Stop Bor: `sudo service bor stop`, or

1. `ps -aux | grep bor`. Get the PID for Bor and then run the following command.
2. `sudo kill -9 PID`

For Binaries:

Stop Heimdall: `pkill heimdalld`

Stop Bridge: `pkill heimdalld-bridge`

Stop Bor: Go to CS-2001/bor and then run, `bash stop.sh`

To remove Heimdall and Bor directories

For Linux packages:

Delete Heimdall: `sudo rm -rf /etc/heimdall/*`

Delete Bor: `sudo rm -rf /etc/bor/*`

For Binaries:

Delete Heimdall: `sudo rm -rf /var/lib/heimdall/`

Delete Bor: `sudo rm -rf /var/lib/bor`

List of common commands

Where to find Heimdall genesis file

`$CONFIGPATH/heimdall/config/genesis.json`

Where to find `heimdall-config.toml`

`/etc/heimdall/config/heimdall-config.toml`

Where to find `config.toml`

`/etc/heimdall/config/config.toml`

`Where to find heimdall-seeds.txt`

`$CONFIGPATH/heimdall/heimdall-seeds.txt`

Start Heimdall

`$ sudo service heimdalld start`

Start Heimdall rest-server

`$ sudo service heimdalld-rest-server start`

Start Heimdall bridge-server

`$ sudo service heimdalld-bridge start`

Heimdall logs

`/var/log/matic-logs/`

Where to find Bor genesis file

`$CONFIGPATH/bor/genesis.json`

Start Bor

`sudo service bor start`

Check Heimdall logs

`tail -f heimdalld.log`

Check Heimdall rest-server

`tail -f heimdalld-rest-server.log`

Check Heimdall bridge logs

`tail -f heimdalld-bridge.log`

Check Bor logs

`tail -f bor.log`

Kill Bor process

For Linux

1. `ps -aux | grep bor`. Get the PID for Bor and then run the following command.
2. `sudo kill -9 PID`

For binaries

Go to `CS-2003/bor` and then run, `bash stop.sh`

Diagnosing what went wrong in a node

You can use [this script]() to check periodially the sync status of your node.

How to Troubleshoot Your Node Mining More Zero Blocks

If your node is mining more zero blocks, it's recommended to compare the size of the chaindata directory on both the sentry and validator. If the sizes are not matching, you should clear the chaindata directory that is less in size, and we recommend syncing from a fresh snapshot again.

Steps to clear chaindata:

1. Run the command `rm -r /var/lib/bor/data/bor/chaindata`. This command removes the chaindata directory and all its contents recursively. This step is necessary to clear any corrupted data that might be causing issues with your node.
2. Then, run `mkdir /var/lib/bor/data/bor/chaindata`. This command creates a new, empty chaindata directory, which will be populated with the latest block data when you start syncing your node again. This step is necessary to ensure that your node has a clean slate to work with, and it won't face any issues caused by previous corrupted data.

To sync from a snapshot, please follow the instructions provided here

How to Address a Significant Block Reorg

If your node has suffered a significant reorg, we recommend that you connect with the Polygon Team and share the logs. Our team is continuously working on improving the network's finality in the long term.

Using Public IP Addresses for `persistent_peers` and `trusted_nodes` Configuration

When configuring the `persistent_peers` and `trusted_nodes` settings, we recommend using public IP addresses. This ensures that your node is connected to the correct public network and can communicate with other nodes effectively. Using private IP addresses can result in connectivity issues, as these addresses are not accessible outside your private network.

How to Resolve Sync Issues for a Falling-Behind Validator

If your validator is falling behind the latest blocks and is currently syncing, follow these steps:

1. Open port `8545` on the sentry.

2. In the heimdall-config.toml file of the validator, replace it with http://ip:8545.

3. Check that the sentry RPC has the http API enabled in config.toml.

4. The validator must have the address of `0.0.0.0` and only listen to and from the validator.

> *Once the node has caught up with the latest blocks, please revert all the changes to the defaults.*

If the sentry has also fallen behind the latest blocks, follow these additional steps:

1. Open the heimdall-config.toml file on the validator.
2. Change the value of the `bor_rpc_url` parameter to any one of the public endpoints listed here.
3. Restart the services.

> *Once the node has caught up with the latest blocks, please revert all the changes to the defaults.*

How to Troubleshoot Errors with Heimdall Bridge Tasks

If you receive an error message similar to the following:

```
[0;94mINFO: 2023/02/24 09:25:12 amqp.go:326 Task not registered with this worker. Requeing message: {"UUID":"task_5c2d8983-89fa-
42c1-a491-bacb682572b0","Name":"sendStakeUpdateToHeimdall","RoutingKey":"machinery_task","ETA":"2023-02-
24T07:25:04.0895409Z","GroupUUID":"","GroupTaskCount":0,"Args":[{"Name":"","Type":"string","Value":"StakeUpdate"},
{"Name":"","Type":"string","Value":"{\"address\":\"0xa59c847bd5ac0172ff4fe912c5d29e5a71a7512b\",\"topics\":
[\"0x35af9eea1f0e7b300b0a14fae90139a072470e44daa3f14b5069bebbc1265bda\",\"0x0000000000000000000000000000000000000000000000000000
{},"Priority":0,"Immutable":false,"RetryCount":2,"RetryTimeout":1,"OnSuccess":null,"OnError":null,"ChordCallback":null,"BrokerMessa
[0m
```

It generally indicates that the Heimdall bridge is not properly up and running. Usually, the issue gets resolved in some time. However, if the problem persists, restarting the node can help resolve it.
--- id: access-node-alchemy title: Access Node with Alchemy description: Make blockchain queries on Polygon without building a full node using the Alchemy SDK. keywords: - docs - matic - polygon - node - alchemy - alchemy sdk - query - full node image: https://wiki.polygon.technology/img/polygon-wiki.png --- import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem'; import useBaseUrl from '@docusaurus/useBaseUrl';

If you're a new Web3 developer, it's unlikely that you'll need to run your own full node on Polygon.

The majority of developers use a node provider, or a third-party external service that receives node requests and returns responses for you automatically. That's because the fastest way to get developing on Polygon is using a node provider rather than managing your own node.

:::tip Recommended

New developers on the Polygon network can use Alchemy as their node provider because of it's reliability, scale, and Enhanced APIs.

:::

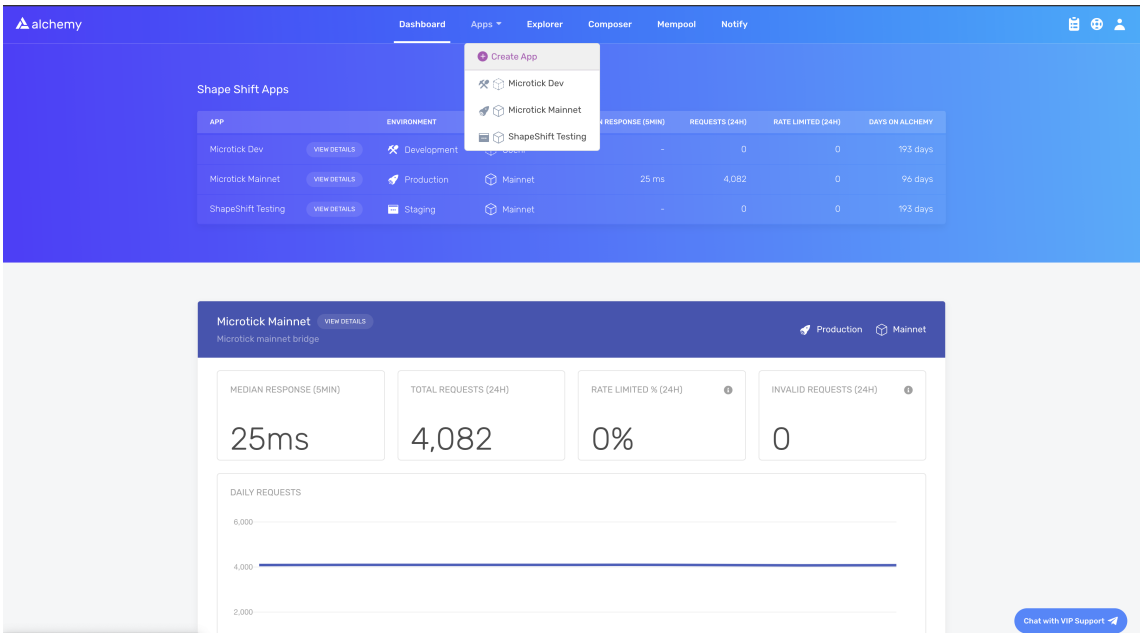
Send Your First Blockchain Request

This guide assumes you already have an Alchemy account and access to the Alchemy Dashboard.

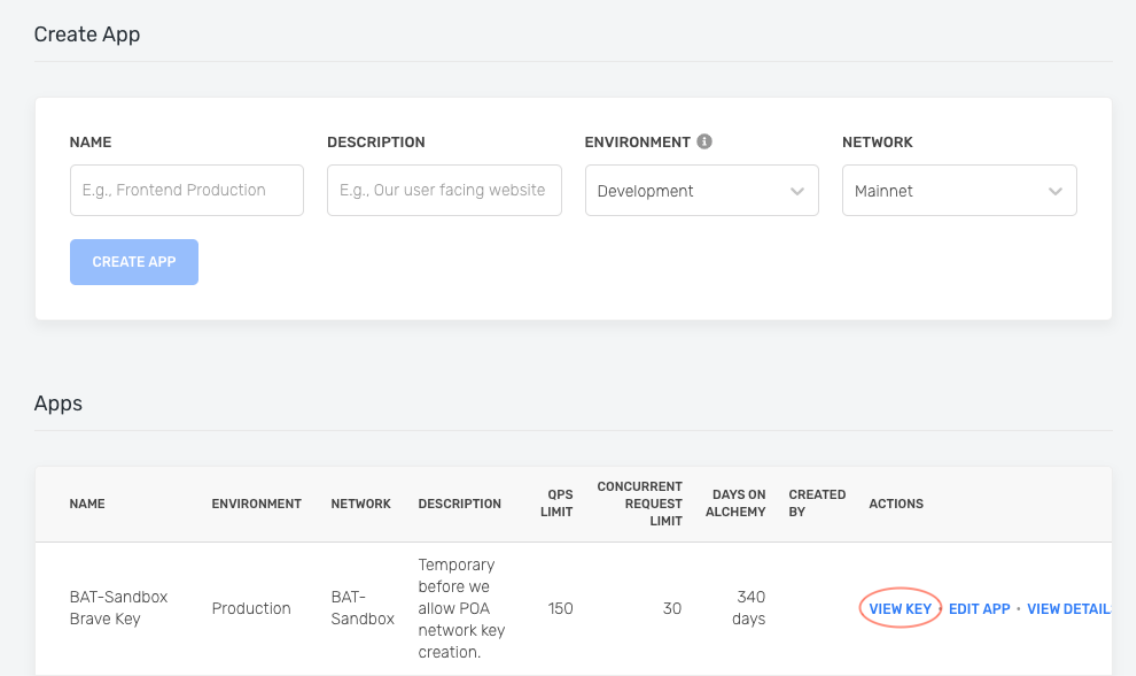
Create an Alchemy Key

First, you'll need an Alchemy API key to authenticate your requests. You can [create API keys from the dashboard]. Check out [this YouTube video] on how to create an app. Or you can follow the steps written below:

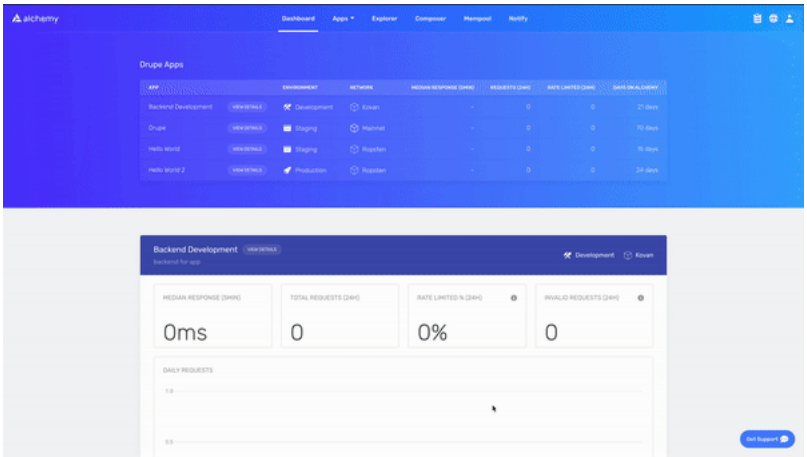
Step 1 → Navigate to the **Create App** button in the **Apps** tab.



Step 2 → Fill in the details under **Create App** to get your new key. You can also see the applications you previously made by you and your team on this page. Pull existing keys by clicking on **View Key** for any app.



Optional → You can also pull existing API keys by hovering over **Apps** and selecting one. You can **View Key** here, as well as **Edit App** to whitelist specific domains, see several developer tools, and view analytics.



Make Your First CURL Request

You can interact with Alchemy's Polygon infrastructure provider using JSON-RPC and your command line interface.

For manual requests, we recommend interacting with the `JSON-RPC` via `POST` requests. Simply pass in the `Content-Type: application/json` header and your query as the `POST` body with the following fields:

- `jsonrpc` : The JSON-RPC version—currently, only `2.0` is supported.
- `method` : The MATIC API method. See API reference.
- `params` : A list of parameters to pass to the method.
- `id` : The ID of your request. Will be returned by the response so you can keep track of which request a response belongs to.

Here is an example you can run from the Terminal/Windows/LINUX command line to retrieve the current gas price:

```
curl https://matic-mainnet.alchemyapi.io/v2/demo \
-X POST \
-H "Content-Type: application/json" \
-d '{"jsonrpc":"2.0","method":"eth_gasPrice","params":[],"id":73}'
```

:::info

In case you want to **send requests to your own app instead of our public demo**;

Replace https://eth-mainnet.alchemyapi.io/jsonrpc/demo with your own API key https://eth-mainnet.alchemyapi.io/v2/your-api-key

:::

Results:

```
{ "id": 73,
  "jsonrpc": "2.0",
  "result": "0x09184e72a000" // 10000000000000 }
```

Set Up the Alchemy SDK

To make blockchain requests directly from your Javascript / Node.js dApp, you'll need to integrate the **Alchemy SDK**, the easiest and most powerful way to access the blockchain and Alchemy's suite of enhanced APIs.

If you have an existing client such as Web3.js or Ethers.js, you can just change your current node provider URL to an Alchemy URL with your API key: https://eth-mainnet.alchemyapi.io/v2/your-api-key

:::note

The scripts below need to be run in a **node context** or **saved in a file**, not run from the command line.

:::

Install the Alchemy SDK

To install the Alchemy SDK, you want to create a project, and then navigate to your project directory to run the installation. Let's go ahead and do that! Once we're in our home directory, let's execute the following:

With Yarn:

```
mkdir your-project-name
cd your-project-name
```

```
yarn init # (or yarn init --yes)
yarn add alchemy-sdk
```

With NPM:

```
mkdir your-project-name
cd your-project-name
npm init   # (or npm init --yes)
npm install alchemy-sdk
```

Create `index.js`

Make sure to replace `demo` with your Alchemy HTTP API key!

```
// Setup: npm install alchemy-sdk
const { Network, Alchemy } = require("alchemy-sdk");

// Optional Config object, but defaults to demo api-key and eth-mainnet.
const settings = {
  apiKey: "demo", // Replace with your Alchemy API Key.
  network: Network.MATIC_MAINNET, // Replace with your network.
};

const alchemy = new Alchemy(settings);

async function main() {
  const latestBlock = await alchemy.core.getBlockNumber();
  console.log("The latest block number is", latestBlock);
}

main();
```

If you are unfamiliar with the async stuff, check out this [Medium post](#).

Run `index.js` Using node

```
node index.js
```

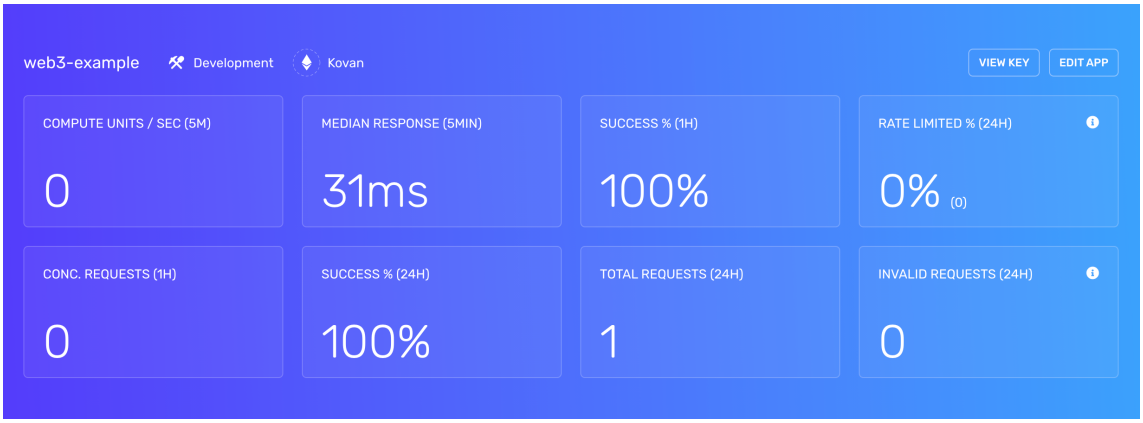
Output in Your Console

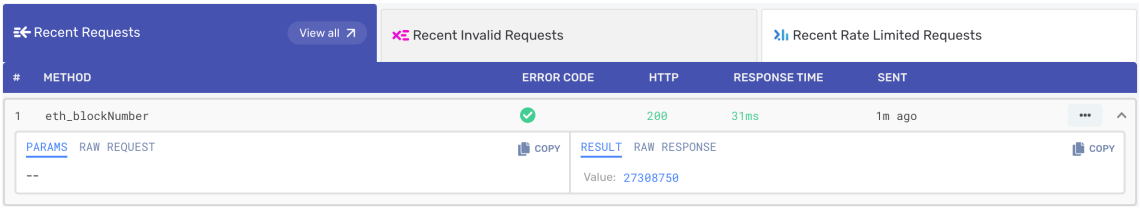
You should now see the latest block number output in your console.

```
The latest block number is 11043912
```

Excellent! You just wrote a working Web3 script and sent your first request to your Alchemy API endpoint.

The project associated with your API key should now look like this on the dashboard:





Start Building

Don't know where to start? Check out these tutorials to get more familiar with Alchemy and blockchain development:

1. Examples of Common Queries Using the Alchemy SDK
2. Learn How to Send Transactions on Ethereum
3. Try deploying your first Hello World Smart Contract and get your hands dirty with some solidity programming!

Other Web3 Libraries

There are a number of alternative Web3 libraries other than the Alchemy SDK you may already be using. See the documentation for these libraries below:

- Web3.py
- Web3j
- Ethers.js
- Web3.js

Using the below code snippets, you can install and use Alchemy as a provider via any of the following libraries.

```
# Setup: pip install web3
from web3 import Web3
alchemy = Web3(Web3.HTTPProvider("https://eth-mainnet.alchemyapi.io/v2/your-api-key"));
```

```
// Setup: curl -L get.web3j.io | sh
Web3j web3 = Web3j.build(new HttpService("https://eth-mainnet.alchemyapi.io/v2/your-api-key"));
```

```
// Setup: npm install ethers
const ethers = require("ethers");
const url = "https://eth-mainnet.alchemyapi.io/v2/your-api-key";
const customHttpProvider = new ethers.providers.JsonRpcProvider(url);
```

```
// Setup: npm install web3
const Web3 = require('web3');
const web3 = new Web3("https://eth-mainnet.alchemyapi.io/v2/your-api-key");
```

---

id: archive-node-nodereal title: Using NodeReal description: Quick and easy third-party service that handles node requests and returns archive responses keywords: - docs - matic - polygon - node - nodereal - query - archive node image: https://wiki.polygon.technology/img/polygon-logo.png

import useBaseUrl from '@docusaurus/useBaseUrl';

:::caution Content disclaimer

Please view the third-party content disclaimer here.

:::

Get access to an Archive Node with NodeReal

If you're new to Web3 development, NodeReal simplifies the process of setting up a full node on Polygon.

:::tip Simplify Polygon node usage with NodeReal and its quality assurance capabilities

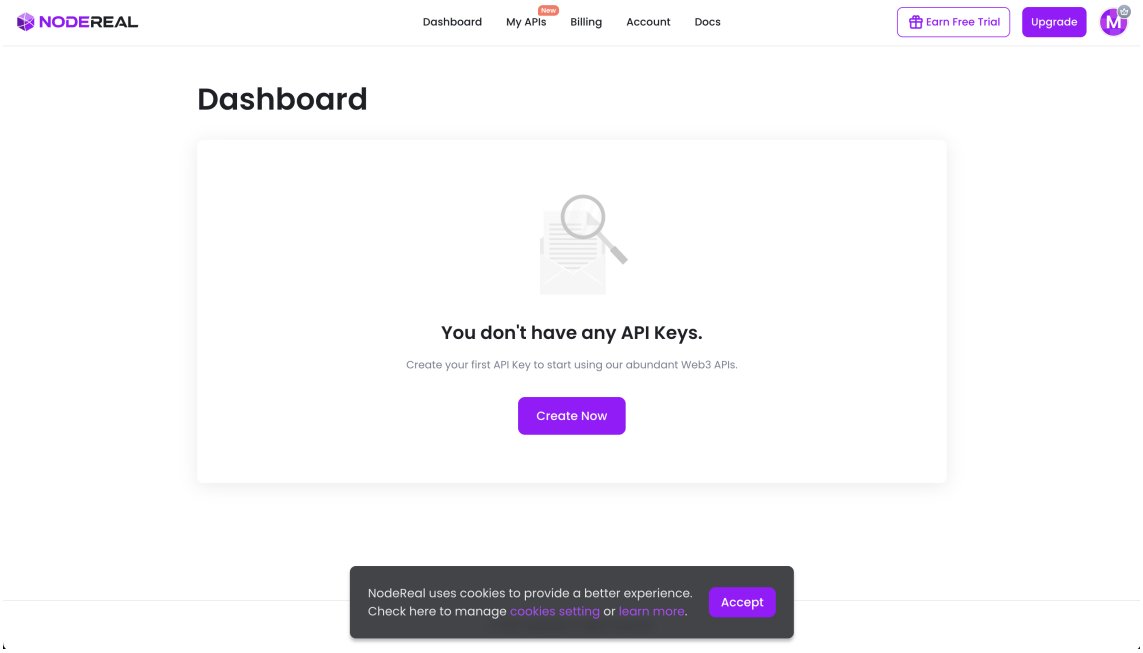
Many developers prefer to use a node provider when getting started on Polygon, or any chain for that matter, as it offers a quick and easy way to handle requests and responses. NodeReal offers a range of features for developers looking for a node provider on the Polygon network. These features include a Standard Interface that simplifies node usage, as well as reliability and scalability for optimal performance. Additionally, NodeReal offers Archive Data for secure data storage, and an API Marketplace that provides additional resources for developers looking to build on Polygon.

:::

Send Your First Blockchain API Request

If you haven't signed up for a MegaNode account yet, you can find a step-by-step guide to getting started here. Once you have a MegaNode account and access to the NodeReal Dashboard, you're ready to proceed.

Create Your First API Key



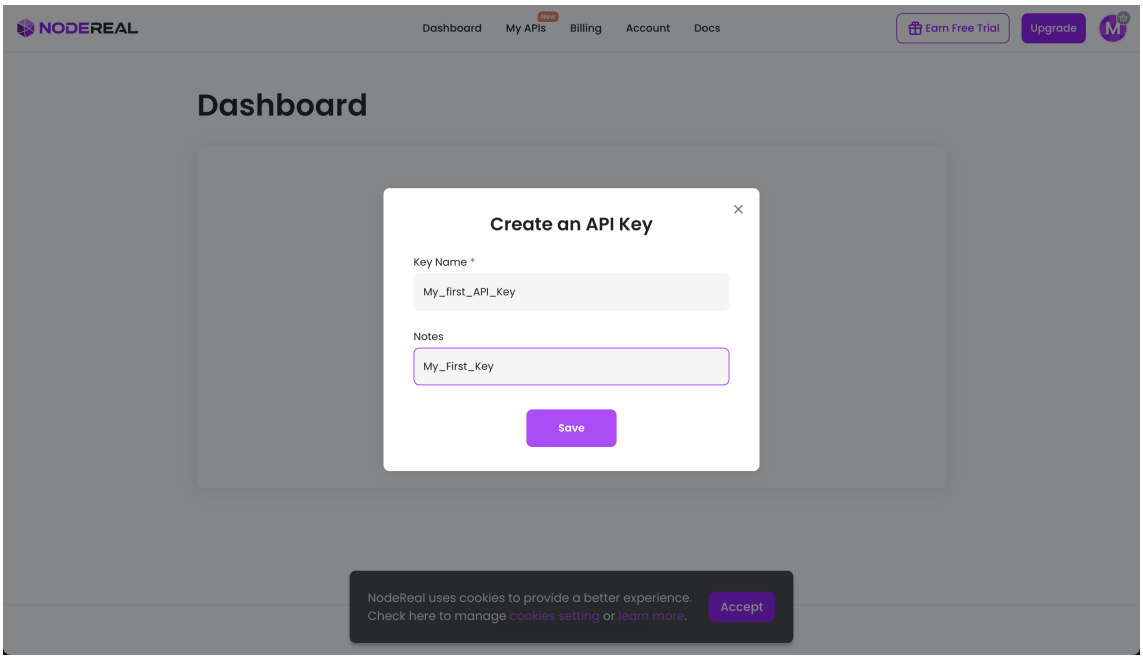
To start sending RPC requests on MegaNode, you'll need to create an API Key. This project-based unit is used to manage your API requests, and requires you to input a name for the key and leave any notes you prefer for future reference. Here's how to get started:

:::tip One API key for all chains

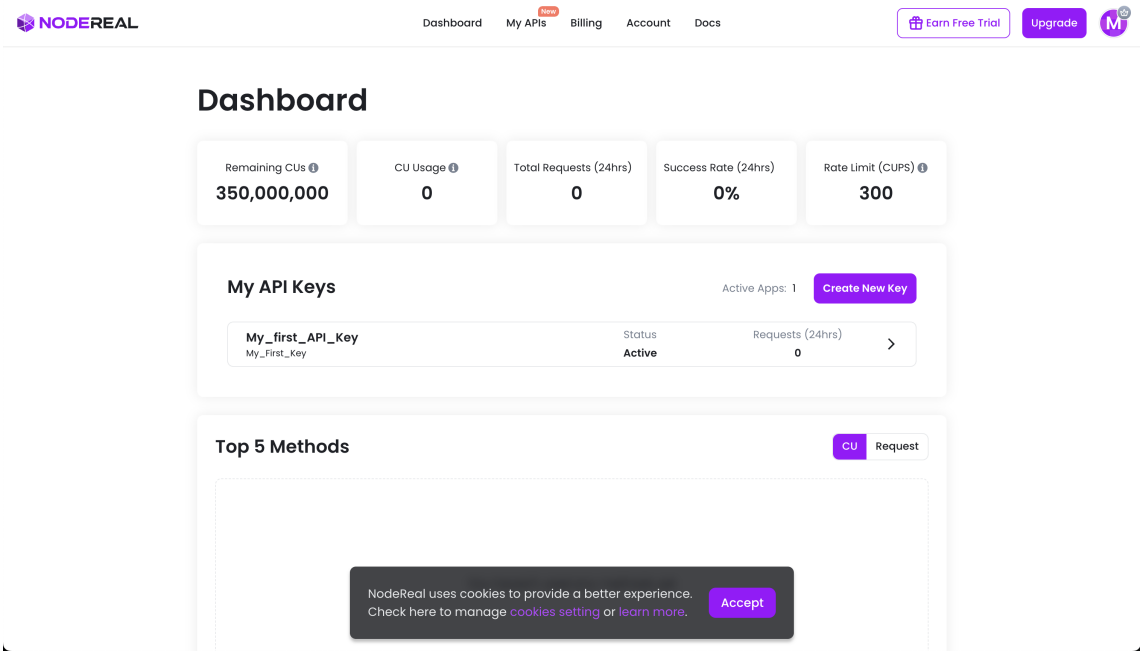
With only one API Key, you can request all the chains and networks we support on NodeReal.

:::

Get started → Input an API **Key name** and leave any **notes** you prefer:



After you have created an API Key, you can find a dashboard on the main screen.



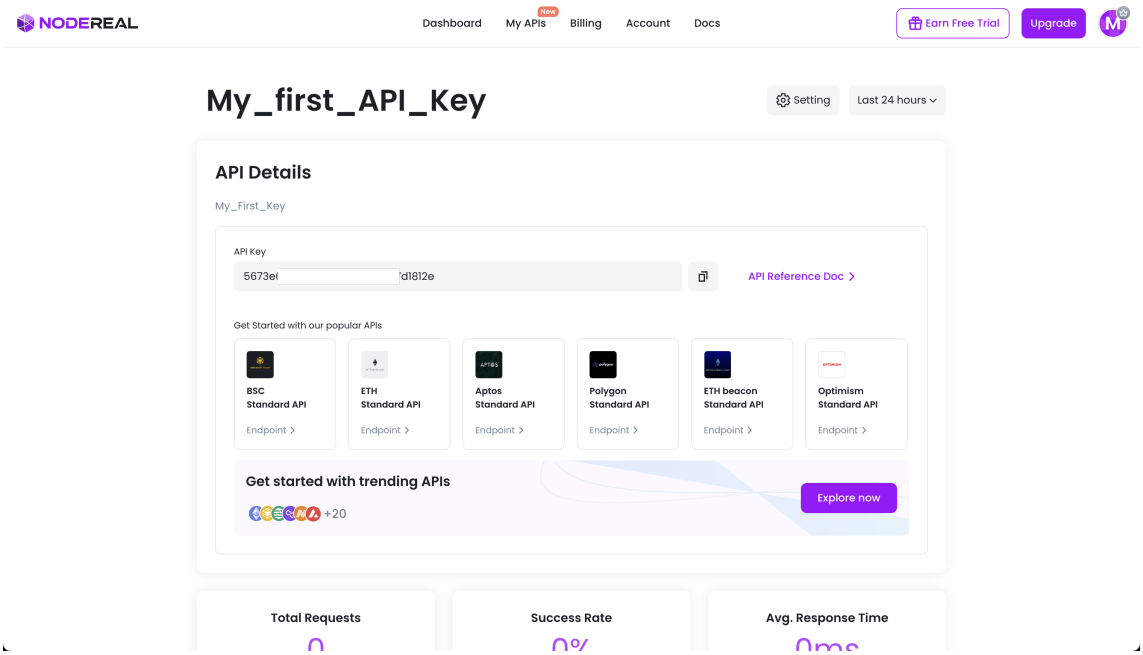
:::info How many API Keys can I create?

The number of available API Keys is subjected to the pricing plan.

:::

Find Your API Key

After you have successfully created an API Key, you can find the API endpoints on the API Key detail page. Please note that this API key is used to identify a particular user and should **NOT** be shared with anyone.



Make Your First RPC Request

After you find the API Key, you can now start to send RPC requests on MegaNode. Please note that all usage will be calculated on an account basis. You can find our API docs here. Furthermore, you can find more information about the Compute Unit (CU) usage here and the account Compute Unit Per Second (CUPS) rate limits here.

:::info Any concern about your Privacy?

Check out NodeReal Privacy Policy.

:::

We'll be using Web3.js libraries to make our first RPC request.

Install the Web3.js SDK

If you want to create a project, install Web3.js SDK, and then navigate to your project directory to run the installation. Once we're in our home directory, let's execute the following:

With Yarn:

```
mkdir your-project-name
cd your-project-name
yarn init # (or yarn init --yes)
yarn add web3
```

With NPM:

```
mkdir your-project-name
cd your-project-name
npm init   # (or npm init --yes)
npm install web3
```

Create `index.js`

Create a JavaScript file named `index.js` and paste the following code:

```
const Web3 = require('web3');

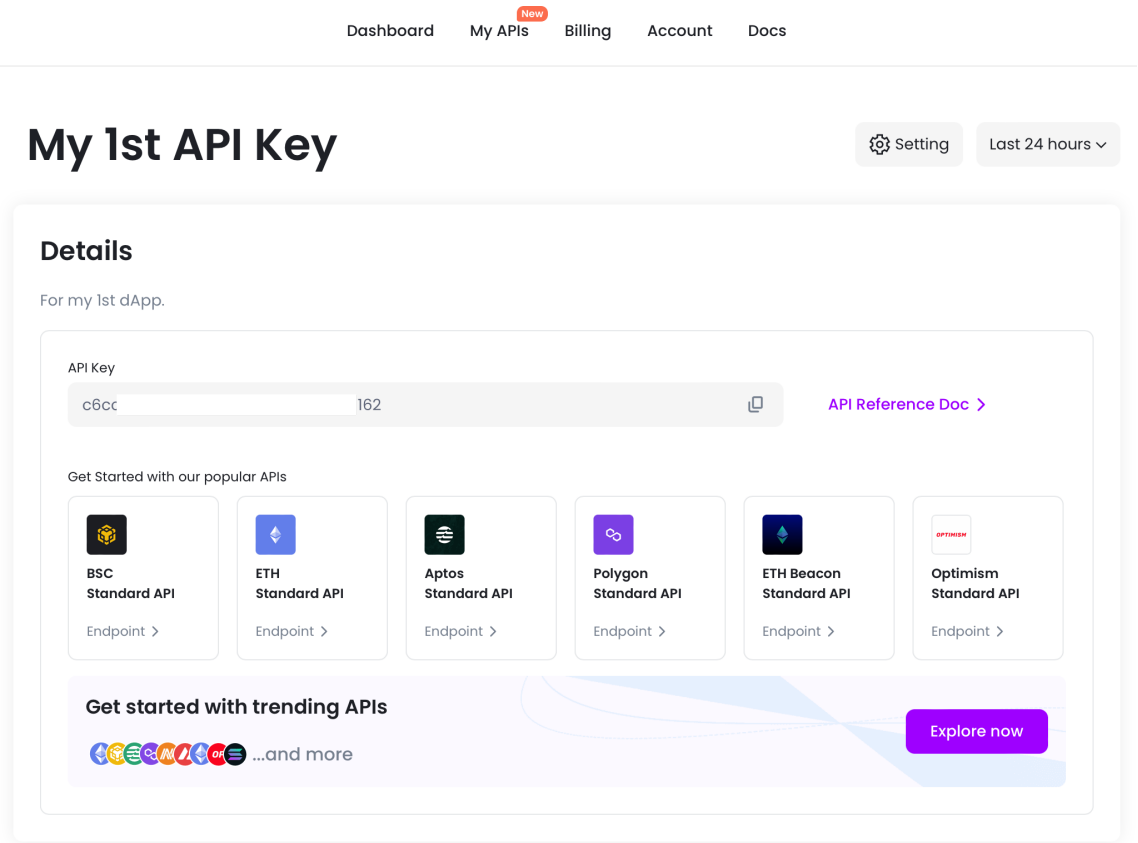
// Set up a new web3 instance
const web3 = new Web3('https://polygon-mainnet.nodereal.io/v1/<API~KEY> ');


// Get the latest block number
web3.eth.getBlockNumber()
  .then((latestBlockNumber) => {
    console.log(`Latest block number: ${latestBlockNumber}`);


    // Get the latest block details
    return web3.eth.getBlock(latestBlockNumber);
  })
  .then((latestBlock) => {
    console.log('Latest block details:', latestBlock);
  })
  .catch((err) => {
    console.error('Error:', err);
  });
```

Make sure to replace `<API KEY>` with your NodeReal Https API key.

You can retrieve the Polygon **HTTPS URL** from your dashboard.



If you encounter any issues, do reach out to us at (Discord)[https://discord.com/invite/nodereal].

Install dependency for `index.js`

npm

```
npm install web3
```

yarn

```
yarn add web3
```

Run `index.js` Using node

```
node index.js
```

Response in Your Terminal

You should be able to see the latest block number and block details on your terminal.

```
Latest block number: 40302452
Latest block details: {............}
```

Self-Service Troubleshooting for MegaNode users

Please refer to MegaNode Errors for more information.

Great work! You managed to write a working Web3 script and sent your first request via NodeReal API endpoint.

id: full-node-deployment title: How to Run a Full PoS Node sidebar_label: Run a Full Node description: "Learn how to deploy a full Polygon PoS node." keywords: - docs - polygon - matic - node - full node setup - ansible - deploy image: https://wiki.polygon.technology/img/polygon-wiki.png

import useBaseUrl from '@docusaurus/useBaseUrl'; import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem';

This deployment guide walks you through starting and running a full node through various methods. For the system requirements, see the Minimum Technical Requirements guide.

:::tip Snapshots

Steps in these guide involve waiting for the Heimdall and Bor services to fully sync. This process takes several days to complete.

Please use snapshots for faster syncing without having to sync over the network. For detailed instructions, see Snapshot Instructions for Heimdall and Bor.

For snapshot download links, see the Polygon Chains Snapshots page.

:::

<Tabs defaultValue="binaries" values={[{ label: 'Binaries', value: 'binaries', }, { label: 'Docker', value: 'docker', }, { label: 'Ansible', value: 'ansible', }, { label: 'Packages', value: 'package', }, { label: 'Google Cloud', value: 'gcp', },] }>

Overview

- Prepare the machine
- Install Heimdall and Bor binaries on the full node machine
- Set up Heimdall and Bor services on the full node machine
- Configure the full node machine
- Start the full node machine
- Check node health with the community

:::note

You have to follow the exact outlined sequence of actions, otherwise you will run into issues.

:::

Install `build-essential`

This is **required** for your full node. In order to install, run the below command:

```
sudo apt-get update
sudo apt-get install build-essential
```

Install Binaries

Polygon node consists of 2 layers: Heimdall and Bor. Heimdall is a tendermint fork that monitors contracts in parallel with the Ethereum network. Bor is basically a Geth fork that generates blocks shuffled by Heimdall nodes.

Both binaries must be installed and run in the correct order to function properly.

Heimdall

Install the latest version of Heimdall and related services. Make sure you checkout to the correct release version. Note that the latest version, Heimdall v0.3.4, contains enhancements such as:

1. Restricting data size in state sync txs to:
   - **30Kb** when represented in **bytes**
   - **60Kb** when represented as **string**
2. Increasing the **delay time** between the contract events of different validators to ensure that the mempool doesn't get filled very quickly in case of a burst of events which can hamper the progress of the chain.

The following example shows how the data size is restricted:

```
Data - "abcd1234"
Length in string format - 8
Hex Byte representation - [171 205 18 52]
Length in byte format - 4
```

To install **Heimdall**, run the below commands:

```
curl -L https://raw.githubusercontent.com/maticnetwork/install/main/heimdall.sh | bash -s -- <heimdall_version> <network_type> <node_type>
```

heimdall_version: `valid v0.3+ release tag from https://github.com/maticnetwork/heimdall/releases` **network_type**: `mainnet` and `mumbai` **node_type**: `sentry`

That will install the `heimdalld` and `heimdallcli` binaries. Verify the installation by checking the Heimdall version on your machine:

```
heimdalld version --long
```

Configure heimdall seeds (Mainnet)

```
sed -i 's|^seeds =.*|seeds =
"2a53a15ffc70ad41b6876ecbe05c50a66af01e20@3.211.248.31:26656,6f829065789e5b156cbbf076f9d133b4d7725847@3.212.183.151:26656,7285a532b
/var/lib/heimdall/config/config.toml
chown heimdall /var/lib/heimdall
```

Configure heimdall seeds (Mumbai)

```
sed -i 's|^seeds =.*|seeds =
"e72c0466a02ea43b2198bd3a9454b87a3ef0d77e@54.147.31.250:26656,b302d1ddb21102e794b524d05152a7834da05bd1@34.226.134.117:26656,9dfc12c
/var/lib/heimdall/config/config.toml
chown heimdall /var/lib/heimdall
```

Bor install

Install the latest version of Bor, based on valid v0.3+ released version.

```
curl -L https://raw.githubusercontent.com/maticnetwork/install/main/bor.sh | bash -s -- <bor_version> <network_type> <node_type>
```

bor_version: `valid v0.3+ release tag from https://github.com/maticnetwork/bor/releases` **network_type**: `mainnet` and `mumbai` **node_type**: `sentry`

That will install the `bor` binary. Verify the installation by checking the Bor version on your machine:

```
bor version
```

Configure bor seeds (mainnet)

```
sed -i 's|.*\[p2p.discovery\]|  \[p2p.discovery\] |g' /var/lib/bor/config.toml
sed -i 's|.*bootnodes =.*|    bootnodes =
["enode://0cb82b395094ee4a2915e9714894627de9ed8498fb881cec6db7c65e8b9a5bd7f2f25cc84e71e89d0947e51c76e85d0847de848c7782b13c0255247a6
/var/lib/bor/config.toml
chown bor /var/lib/bor
```

Configure bor seeds (mumbai)

```
sed -i 's|.*\[p2p.discovery\]|  \[p2p.discovery\] |g' /var/lib/bor/config.toml
sed -i 's|.*bootnodes =.*|    bootnodes =
["enode://320553cda00dfc003f499a3ce9598029f364fbb3ed1222fdc20a94d97dcc4d8ba0cd0bfa996579dcc6d17a534741fb0a5da303a90579431259150de66
/var/lib/bor/config.toml
chown bor /var/lib/bor
```

Update service config user permission

```
sed -i 's/User=heimdall/User=root/g' /lib/systemd/system/heimdalld.service
sed -i 's/User=bor/User=root/g' /lib/systemd/system/bor.service
```

Start Services

Run the full Heimdall node with these commands on your Sentry Node:

```
sudo service heimdalld start
```

Now, you need to make sure that **Heimdall is synced** completely, and then only start Bor. If you start Bor without Heimdall syncing completely, you will run into issues frequently.

To check if Heimdall is synced

1. On the remote machine/VM, run `curl localhost:26657/status`
2. In the output, `catching_up` value should be `false`

Once Heimdall is synced, run the below command:

```
sudo service bor start
```

Logs

Logs can be managed by the `journalctl` linux tool. Here is a tutorial for advanced usage: How To Use Journalctl to View and Manipulate Systemd Logs.

Check Heimdall node logs

```
journalctl -u heimdalld.service -f
```

Check Heimdall Rest-server logs

```
journalctl -u heimdalld-rest-server.service -f
```

Check Bor Rest-server logs

```
journalctl -u bor.service -f
```

Ports and Firewall Setup

Open ports 22, 26656 and 30303 to world (0.0.0.0/0) on sentry node firewall.

You can use VPN to restrict access for port 22 as per your requirement and security guidelines.
The Polygon team distributes official Docker images which can be used to run nodes on the Polygon Mainnet. These instructions are for running a Full Node, but they can be adapted for running sentry nodes and validators as well.

Prerequisites

The general configuration for running a Polygon full node is to have **at least** 4 CPUs/cores and 16 GB of RAM. For this walk through, we're going to be using AWS and a `t3.2xlarge` instance type. The application can run on both x86 and ARM architectures.

These instructions are based on Docker, so it should be easy to follow along with almost any operating system, but we're using Ubuntu.

In terms of space, for a full node you'll probably need from **2.5 to 5 terabytes of SSD (or faster) storage**.

The peer exchange for a Polygon full node generally depends on port 30303 and 26656 being open. When you configure your firewall or security groups for AWS, make sure these ports are open along with whatever ports you need to access the machine.

TLDR:

- Use a machine with at least 4 cores and 16GB RAM
- Make sure you have from 2.5 TB to 5 TB of fast storage
- Use a public IP and open ports 30303 and 26656

Initial Setup

At this point, you should have shell access with root privileges to a linux machine.

img

Install Docker

Most likely your operating system won't have Docker installed by default. Please follow the instructions for your particular distribution found here: https://docs.docker.com/engine/install/

We're following the instructions for Ubuntu. The steps are included below, but please see the official instructions in case they've been updated.

```
sudo apt-get update
sudo apt-get install ca-certificates curl gnupg lsb-release
sudo mkdir -p /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-compose-plugin
```

At this point you should have Docker installed. In order to verify, you should be able to run a command like this:

```
sudo docker run hello-world
```

img

In many cases, it's inconvenient to run docker as `root` user so we'll follow the post install steps here in order to interact with docker without needing to be `root` :

```
sudo groupadd docker
sudo usermod -aG docker $USER
```

Now you should be able to logout and log back in and run docker commands without `sudo` .

Disk Setup

The exact steps required here are going to vary a lot based on your needs. Most likely you'll have a root partition running your operating system on one device. You'll probably want one or more devices for actually holding the blockchain data. For the rest of the walkthrough, we're going to have that additional device mounted at `/mnt/data`.

In this example, we have a device with 4 TB of available space located at `/dev/nvme1n1`. We are going to mount that using the steps below:

```
sudo mkdir /mnt/data
sudo mount /dev/nvme1n1 /mnt/data
```

We use `df -h` to make sure the mount looks good.

img

If that all looks good, we might as well create the home directories on this mount for Bor and Heimdall.

```
sudo mkdir /mnt/data/bor
sudo mkdir /mnt/data/heimdall
```

Depending on your use case and operating system, you'll likely want to create an entry in `/etc/fstab` in order to make sure your device is mounted when the system reboots.

In our case we're following some steps like this:

```
# Use blkid to get the UUID for the device that we're mounting
blkid

# Edit the fstab file  and add a line to mount your device
# UUID={your uuid}              /mnt/data       {your filesystem}       defaults        0       1
sudo emacs /etc/fstab

# use this to verify the fstab actually works
sudo findmnt --verify --verbose
```

At this point you should be able to reboot and confirm that the system loads your mount properly.

Heimdall Setup

At this point, we have a host with docker running on it and we have ample mounted storage to run our Polygon node software. So let's get Heimdall configured and running.

First let's make sure we can run Heimdall with docker. Run the following command:

```
docker run -it 0xpolygon/heimdall:0.3.4 heimdallcli version
```

If this is the first time you've run Heimdall with docker, it should pull the required image automatically and output the version information.

img

If you'd like to check the details of the Heimdall image or find a different tag, you can take a look at the repository on Docker Hub: https://hub.docker.com/repository/docker/0xpolygon/heimdall

At this point, let's run the Heimdall `init` command to set up our home directory.

```
docker run -v /mnt/data/heimdall:/heimdall-home:rw --entrypoint /usr/bin/heimdalld -it 0xpolygon/heimdall:0.3.4 init --
home=/heimdall-home
```

Let's break this command down a bit in case anything goes wrong.

- We're using `docker run` to run a command via docker.
- The switch `-v /mnt/data/heimdall:/heimdall-home:rw` is very important. It's mounting the folder that we created earlier `/mnt/data/heimdall` from our host system to `/heimdall-home` within the container as a docker volume.
- The `rw` allows the command to write to this docker volume. For all intents and purposes, from within the docker container, the home directory for Heimdall will be `/heimdall-home`.
- The argument `--entrypoint /usr/bin/heimdalld` is overriding the default entry point for this container.
- The switch `-it` is used to run the command interactively.
- Finally we're specifying which image we want to run with `0xpolygon/heimdall:0.3.4`.
- After that `init --home=/heimdall-home` are arguments being passed to the heimdalld executable. `init` is the command we want to run and `--home` is used to specify the location of the home directory.

After running the `init` command, your `/mnt/data/heimdall` directory should have some structure and look like this:

img

Now we need to make a few updates before starting Heimdall. First we're going to edit the `config.toml` file.

```
# Open the config.toml and and make three edits
# moniker = "YOUR NODE NAME HERE"
# laddr = "tcp://0.0.0.0:26657"
# seeds = "LATEST LIST OF SEEDS"

sudo emacs /mnt/data/heimdall/config/config.toml
```

If you don't have a list of seeds, you can find one in this section. In our case, our file has these three lines:

```
# A custom human readable name for this node
moniker="examplenode01"

# TCP or UNIX socket address for the RPC server to listen on
laddr = "tcp://0.0.0.0:26657"

# Comma separated list of seed nodes to connect to
seeds="f4f605d60b8ffaaf15240564e58a81103510631c@159.203.9.164:26656,4fb1bc820088764a564d4f66bba1963d47d82329@44.232.55.71:26656,2eac
```

:::caution

There are two `laddr` inside `config.toml` file. Make sure that you only change the `laddr` parameter under `[rpc]` section.

:::

Now that your `config.toml` file is all set, you'll need to make two small changes to your `heimdall-config.toml` file. Use your favorite editor to update these two settings:

```
# RPC endpoint for ethereum chain
eth_rpc_url = "http://localhost:9545"

# RPC endpoint for bor chain
bor_rpc_url = "http://localhost:8545"
```

The `eth_rpc_url` should be updated to whatever URL you use for Ethereum Mainnet RPC. The `bor_rpc_url` in our case is going to be updated to `http://bor:8545`. After making the edits, our file has these lines:

```
# RPC endpoint for ethereum chain
eth_rpc_url = "https://eth-mainnet.g.alchemy.com/v2/ydmGjsREDACTED_DONT_USE9t7FSf"

# RPC endpoint for bor chain
bor_rpc_url = "http://bor:8545"
```

The default `init` command provides a `genesis.json` but that will not work with Polygon Mainnet or Mumbai. If you're setting up a mainnet node, you can run this command to download the correct genesis file:

```
sudo curl -o /mnt/data/heimdall/config/genesis.json
https://raw.githubusercontent.com/maticnetwork/heimdall/master/builder/files/genesis-mainnet-v1.json
```

If you want to verify that you have the right file, you can check against this hash:

```
# sha256sum genesis.json
498669113c72864002c101f65cd30b9d6b159ea2ed4de24169f1c6de5bcccf14  genesis.json
```

Starting Heimdall

Before we start Heimdall, we're going to create a docker network so that the containers can easily network with each other based on names. In order to create the network, run the following command:

```
docker network create polygon
```

Now we're going to start Heimdall. Run the following command:

```
docker run -p 26657:26657 -p 26656:26656 -v /mnt/data/heimdall:/heimdall-home:rw --net polygon --name heimdall --entrypoint
/usr/bin/heimdalld -d --restart unless-stopped  0xpolygon/heimdall:0.3.4 start --home=/heimdall-home
```

Many of the pieces of this command will look familiar. So let's talk about what's new.

- The `-p 26657:26657` and `-p 26656:26656` switches are port mappings. This will instruct docker to map the host port `26657` to the container port `26657` and the same for `26656`.
- The `--net polygon` switch is telling docker to run this container in the polygon network.

- `--name heimdall` is naming the container which is useful for debugging, but it's all the name that will be used for other containers to connect to Heimdall.
- The `-d` argument tells docker to run this container in the background.
- The switch `--restart unless-stopped` tells docker to automatically restart the container unless it was stopped manually.
- Finally, `start` is being used to actually run the application instead of `init` which just set up the home directory.

At this point it's helpful to check and see what's going on. These two commands can be useful:

```
# ps will list the running docker processes. At this point you should see one container running
docker ps

# This command will print out the logs directly from the heimdall application
docker logs -ft heimdall
```

At this point, Heimdall should start syncing. When you look at the logs, you should see a log of information being spit out that looks like this:

```
2022-12-14T19:43:23.687640820Z INFO [2022-12-14|19:43:23.687] Executed block                          module=state
height=26079 validTxs=0 invalidTxs=0
2022-12-14T19:43:23.721220869Z INFO [2022-12-14|19:43:23.721] Committed state                          module=state
height=26079 txs=0 appHash=CAEC4C181C9F82D7F55C4BB8A7F564D69A41295A3B62DDAA45F2BB41333DC20F
2022-12-14T19:43:23.730533414Z INFO [2022-12-14|19:43:23.730] Executed block                          module=state
height=26080 validTxs=0 invalidTxs=0
2022-12-14T19:43:23.756646938Z INFO [2022-12-14|19:43:23.756] Committed state                          module=state
height=26080 txs=0 appHash=CAEC4C181C9F82D7F55C4BB8A7F564D69A41295A3B62DDAA45F2BB41333DC20F
2022-12-14T19:43:23.768129711Z INFO [2022-12-14|19:43:23.767] Executed block                          module=state
height=26081 validTxs=0 invalidTxs=0
2022-12-14T19:43:23.794323918Z INFO [2022-12-14|19:43:23.794] Committed state                          module=state
height=26081 txs=0 appHash=CAEC4C181C9F82D7F55C4BB8A7F564D69A41295A3B62DDAA45F2BB41333DC20F
2022-12-14T19:43:23.802989809Z INFO [2022-12-14|19:43:23.802] Executed block                          module=state
height=26082 validTxs=0 invalidTxs=0
2022-12-14T19:43:23.830960386Z INFO [2022-12-14|19:43:23.830] Committed state                          module=state
height=26082 txs=0 appHash=CAEC4C181C9F82D7F55C4BB8A7F564D69A41295A3B62DDAA45F2BB41333DC20F
2022-12-14T19:43:23.840941976Z INFO [2022-12-14|19:43:23.840] Executed block                          module=state
height=26083 validTxs=0 invalidTxs=0
2022-12-14T19:43:23.866564767Z INFO [2022-12-14|19:43:23.866] Committed state                          module=state
height=26083 txs=0 appHash=CAEC4C181C9F82D7F55C4BB8A7F564D69A41295A3B62DDAA45F2BB41333DC20F
2022-12-14T19:43:23.875395744Z INFO [2022-12-14|19:43:23.875] Executed block                          module=state
height=26084 validTxs=0 invalidTxs=0
```

If you're not seeing any information like this, your node might not be finding enough peers. The other useful command at this point is an RPC call to check the status of Heimdall syncing:

```
curl localhost:26657/status
```

This will return a response like:


```
{
  "jsonrpc": "2.0",
  "id": "",
  "result": {
    "node_info": {
      "protocol_version": {
        "p2p": "7",
        "block": "10",
        "app": "0"
      },
      "id": "0698e2f205de0ffbe4ca215e19b2ee7275d2c334",
      "listen_addr": "tcp://0.0.0.0:26656",
      "network": "heimdall-137",
      "version": "0.32.7",
      "channels": "4020212223303800",
      "moniker": "examplenode01",
      "other": {
        "tx_index": "on",
        "rpc_address": "tcp://0.0.0.0:26657"
      }
    },
    "sync_info": {
      "latest_block_hash": "812700055F33B175CF90C870B740D01B0C5B5DCB8D22376D2954E1859AF30458",
      "latest_app_hash": "83A1568E85A1D942D37FE5415F3FB3CBD9DFD846A42CBC247DFD6ABB9CE7E606",
      "latest_block_height": "16130",
      "latest_block_time": "2020-05-31T17:06:31.350723885Z",
      "catching_up": true
    },
```

```
    "validator_info": {
      "address": "3C6058AF387BB74D574582C2BEEF377E7A4C0238",
      "pub_key": {
        "type": "tendermint/PubKeySecp256k1",
        "value": "BOIKA6z1q3l5iSJoaAiagWpwUw3taAhiEMyZ9ffxAMznas2GU1giD5YmtnrB6jzp4kkIqv4tOmuGYILSdy9+wYI="
      },
      "voting_power": "0"
    }
  }
}
```


In this initial setup phase, it's important to pay attention to the `sync_info` field. If `catching_up` is true, it means that Heimdall is not fully synced. You can check the other properties within `sync_info` to get a sense how far behind Heimdall is.

Starting Bor

At this point, you should have a node that's successfully running Heimdall. You should be ready now to run Bor.

Before we get started with Bor, we need to run the Heimdall rest server. This command will start a REST API that Bor uses to retrieve information from Heimdall. The command to start server is:

```
docker run -p 1317:1317 -v /mnt/data/heimdall:/heimdall-home:rw --net polygon --name heimdallrest --entrypoint
/usr/bin/heimdalld -d --restart unless-stopped 0xpolygon/heimdall:0.3.4 rest-server --home=/heimdall-home --node
"tcp://heimdall:26657"
```

There are two pieces of this command that are different and worth noting. Rather than running the `start` command, we're running the `rest-server` command. Also, we're passing `~-node "tcp://heimdall:26657"~` which tells the rest server how to communicate with Heimdall.

If this command runs successfully, when you run `docker ps`, you should see two commands containers running now. Additionally, if you run this command you should see some basic output:

```
curl localhost:1317/bor/span/1
```

Bor will rely on this interface. So if you don't see JSON output, there is something wrong!

Now let's download the `genesis` file for Bor specifically:

```
sudo curl -o /mnt/data/bor/genesis.json 'https://raw.githubusercontent.com/maticnetwork/bor/master/builder/files/genesis-
mainnet-v1.json'
```

Let's verify the `sha256 sum` again for this file:

```
# sha256sum genesis.json
4bacbfbe72f0d966412bb2c19b093f34c0a1bd4bb8506629eba1c9ca8c69c778  genesis.json
```

Now we need to create a default config file for starting Bor.

```
docker run -it  0xpolygon/bor:0.4.0 dumpconfig | sudo tee /mnt/data/bor/config.toml
```

This command is going to generate a .toml file with default settings. We're going to make a few changes to the file, so open it up with your favorite editor and make a few updates. Note: we're only showing the lines that are changed.

For reference, you can see the details for the Bor image here: https://hub.docker.com/repository/docker/0xpolygon/bor

```
# Similar to moniker, you might want to update this with a name of your own choosing
identity = "docker.example"

# Setting this to the location of a mount that we'll make
datadir = "/bor-home"

# We'll want to specify some boot nodes
[p2p]
  [pep.discovery]
    bootnodes =
["enode://0cb82b395094ee4a2915e9714894627de9ed8498fb881cec6db7c65e8b9a5bd7f2f25cc84e71e89d0947e51c76e85d0847de848c7782b13c0255247a
"enode://88116f4295f5a31538ae409e4d44ad40d22e44ee9342869e7d68bdec55b0f83c1530355ce8b41fbec0928a7d75a5745d528450d30aec92066ab6ba1ee

# Because we're running inside docker, we'll likely need to change the way we connect to heimdall
[heimdall]
  url = "http://heimdallrest:1317"

# Assumming you want to access the RPC, you'll need to make a change here as well
[jsonrpc]
  [jsonrpc.http]
```

```
    enabled = true
    host = "0.0.0.0"
```

At this point, we should be ready to start Bor. We're going to use this command:

```
docker run -p 30303:30303 -p 8545:8545 -v /mnt/data/bor:/bor-home:rw --net polygon --name bor -d --restart unless-stopped
0xpolygon/bor:0.4.0 server --config /bor-home/config.toml
```

If everything went well, you should see lots of logs that look like this:

```
2022-12-14T19:53:51.989897291Z INFO [12-14|19:53:51.989] Fetching state updates from Heimdall     fromID=4 to=2020-05-
30T23:47:46Z
2022-12-14T19:53:51.989925064Z INFO [12-14|19:53:51.989] Fetching state sync events               queryParams="from-id=4&to-
time=1590882466&limit=50"
2022-12-14T19:53:51.997640841Z INFO [12-14|19:53:51.997] StateSyncData                            Gas=0       Block-number=12800
LastStateID=3 TotalRecords=0
2022-12-14T19:53:52.021990622Z INFO [12-14|19:53:52.021] Fetching state updates from Heimdall     fromID=4 to=2020-05-
30T23:49:58Z
2022-12-14T19:53:52.022015930Z INFO [12-14|19:53:52.021] Fetching state sync events               queryParams="from-id=4&to-
time=1590882598&limit=50"
2022-12-14T19:53:52.040660857Z INFO [12-14|19:53:52.040] StateSyncData                            Gas=0       Block-number=12864
LastStateID=3 TotalRecords=0
2022-12-14T19:53:52.064795784Z INFO [12-14|19:53:52.064] Fetching state updates from Heimdall     fromID=4 to=2020-05-
30T23:52:10Z
2022-12-14T19:53:52.064828634Z INFO [12-14|19:53:52.064] Fetching state sync events               queryParams="from-id=4&to-
time=1590882730&limit=50"
2022-12-14T19:53:52.085029612Z INFO [12-14|19:53:52.084] StateSyncData                            Gas=0       Block-number=12928
LastStateID=3 TotalRecords=0
2022-12-14T19:53:52.132067703Z INFO [12-14|19:53:52.131] ✅ Committing new span                    id=3
startBlock=13056 endBlock=19455
validatorBytes=f8b6d906822710940375b2fc7140977c9c76d45421564e354ed42277d9078227109442eefcda06ead475cde3731b8eb138e88cd0bac3d901823
producerBytes=f8b6d906822710940375b2fc7140977c9c76d45421564e354ed42277d9078227109442eefcda06ead475cde3731b8eb138e88cd0bac3d9018238
2022-12-14T19:53:52.133545235Z INFO [12-14|19:53:52.133] Fetching state updates from Heimdall     fromID=4 to=2020-05-
30T23:54:22Z
2022-12-14T19:53:52.133578948Z INFO [12-14|19:53:52.133] Fetching state sync events               queryParams="from-id=4&to-
time=1590882862&limit=50"
2022-12-14T19:53:52.135049605Z INFO [12-14|19:53:52.134] StateSyncData                            Gas=0       Block-number=12992
LastStateID=3 TotalRecords=0
2022-12-14T19:53:52.152067646Z INFO [12-14|19:53:52.151] Fetching state updates from Heimdall     fromID=4 to=2020-05-
30T23:56:34Z
2022-12-14T19:53:52.152198357Z INFO [12-14|19:53:52.151] Fetching state sync events               queryParams="from-id=4&to-
time=1590882994&limit=50"
2022-12-14T19:53:52.176617455Z INFO [12-14|19:53:52.176] StateSyncData                            Gas=0       Block-number=13056
LastStateID=3 TotalRecords=0
2022-12-14T19:53:52.191060112Z INFO [12-14|19:53:52.190] Fetching state updates from Heimdall     fromID=4 to=2020-05-
30T23:58:46Z
2022-12-14T19:53:52.191083740Z INFO [12-14|19:53:52.190] Fetching state sync events               queryParams="from-id=4&to-
time=1590883126&limit=50"
2022-12-14T19:53:52.223836639Z INFO [12-14|19:53:52.223] StateSyncData                            Gas=0       Block-number=13120
LastStateID=3 TotalRecords=0
2022-12-14T19:53:52.236025906Z INFO [12-14|19:53:52.235] Fetching state updates from Heimdall     fromID=4 to=2020-05-
31T00:00:58Z
2022-12-14T19:53:52.236053406Z INFO [12-14|19:53:52.235] Fetching state sync events               queryParams="from-id=4&to-
time=1590883258&limit=50"
2022-12-14T19:53:52.269611566Z INFO [12-14|19:53:52.269] StateSyncData                            Gas=0       Block-number=13184
LastStateID=3 TotalRecords=0
2022-12-14T19:53:52.283199351Z INFO [12-14|19:53:52.283] Fetching state updates from Heimdall     fromID=4 to=2020-05-
31T00:03:10Z
2022-12-14T19:53:52.283737573Z INFO [12-14|19:53:52.283] Fetching state sync events               queryParams="from-id=4&to-
time=1590883390&limit=50"
2022-12-14T19:53:52.314141359Z INFO [12-14|19:53:52.314] StateSyncData                            Gas=0       Block-number=13248
LastStateID=3 TotalRecords=0
2022-12-14T19:53:52.325150782Z INFO [12-14|19:53:52.325] Fetching state updates from Heimdall     fromID=4 to=2020-05-
31T00:05:22Z
2022-12-14T19:53:52.325171075Z INFO [12-14|19:53:52.325] Fetching state sync events               queryParams="from-id=4&to-
time=1590883522&limit=50"
2022-12-14T19:53:52.354470271Z INFO [12-14|19:53:52.354] StateSyncData                            Gas=0       Block-number=13312
LastStateID=3 TotalRecords=0
2022-12-14T19:53:52.372354857Z INFO [12-14|19:53:52.372] Fetching state updates from Heimdall     fromID=4 to=2020-05-
31T00:07:34Z
2022-12-14T19:53:52.372389214Z INFO [12-14|19:53:52.372] Fetching state sync events               queryParams="from-id=4&to-
time=1590883654&limit=50"
2022-12-14T19:53:52.398246950Z INFO [12-14|19:53:52.398] StateSyncData                            Gas=0       Block-number=13376
LastStateID=3 TotalRecords=0
2022-12-14T19:53:52.413321099Z INFO [12-14|19:53:52.413] Fetching state updates from Heimdall     fromID=4 to=2020-05-
31T00:09:46Z
2022-12-14T19:53:52.413345355Z INFO [12-14|19:53:52.413] Fetching state sync events               queryParams="from-id=4&to-
```

```
time=1590883786&limit=50"
2022-12-14T19:53:52.437176855Z INFO [12-14|19:53:52.437] StateSyncData                          Gas=0       Block-number=13440
LastStateID=3 TotalRecords=0
2022-12-14T19:53:52.450356966Z INFO [12-14|19:53:52.450] Fetching state updates from Heimdall     fromID=4 to=2020-05-
31T00:11:58Z
```

There are a few ways to check the sync state of Bor. The simplest is with `curl`:

```
curl 'localhost:8545/' \
--header 'Content-Type: application/json' \
-d '{
    "jsonrpc":"2.0",
    "method":"eth_syncing",
    "params":[],
    "id":1
}'
```

When you run this command, it will give you a result like:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "currentBlock": "0x2eebf",
    "healedBytecodeBytes": "0x0",
    "healedBytecodes": "0x0",
    "healedTrienodeBytes": "0x0",
    "healedTrienodes": "0x0",
    "healingBytecode": "0x0",
    "healingTrienodes": "0x0",
    "highestBlock": "0x1d4ee3e",
    "startingBlock": "0x0",
    "syncedAccountBytes": "0x0",
    "syncedAccounts": "0x0",
    "syncedBytecodeBytes": "0x0",
    "syncedBytecodes": "0x0",
    "syncedStorage": "0x0",
    "syncedStorageBytes": "0x0"
  }
}
```

This will indicate the `currentBlock` that's been synced and also the `highestBlock` that we're aware of. If the node is already synced, we should get `false`.

Seed nodes and bootnodes

- Heimdall seed nodes:

```
moniker=<enter unique identifier>

# Mainnet:
seeds="2a53a15ffc70ad41b6876ecbe05c50a66af01e20@3.211.248.31:26656,6f829065789e5b156cbbf076f9d133b4d7725847@3.212.183.151:26

# Testnet:
seeds="e72c0466a02ea43b2198bd3a9454b87a3ef0d77e@54.147.31.250:26656,b302d1ddb21102e794b524d05152a7834da05bd1@34.226.134.117:
```

- Bootnodes:

```
# Mainnet:
bootnode
["enode://0cb82b395094ee4a2915e9714894627de9ed8498fb881cec6db7c65e8b9a5bd7f2f25cc84e71e89d0947e51c76e85d0847de848c7782b13c02

# Testnet:
bootnodes
["enode://320553cda00dfc003f499a3ce9598029f364fbb3ed1222fdc20a94d97dcc4d8ba0cd0bfa996579dcc6d17a534741fb0a5da303a90579431259
```

An Ansible playbook is used to configure and manage a full node. See the Minimum Technical Requirements guide for the system requirements.

Prerequisites

- Install Ansible on your local machine with Python3.x. The setup will not work if you have Python2.x.
  - To install Ansible with Python 3.x, you can use pip. If you do not have pip on your machine, follow the steps outlined here. Run `pip3 install ansible` to install Ansible.
- Check the Polygon PoS Ansible repository for requirements.

- You will also need to ensure that Go is **not installed** in your environment. You will run into issues if you attempt to set up your full node through Ansible with Go installed as Ansible requires specific packages of Go to be installed.
- You will also need to make sure that your VM / Machine does not have any previous setups for Polygon Validator or Heimdall or Bor. You will need to delete them as your setup will run into issues.

:::info Heimdall source enhancements

The latest Heimdall version, **v0.3.4**, contains a few enhancements. The delay time between the contract events of different validators **has been increased** to ensure that the mempool doesn't get filled quickly in case of a burst of events that could hamper the chain's progress.

Additionally, the data size **has been restricted in state sync txs to 30Kb (when represented in bytes) and 60Kb (when defined as string)**. For example:

```
Data - "abcd1234"
Length in string format - 8
Hex Byte representation - [171 205 18 52]
Length in byte format - 4
```

:::

Full node setup

- Ensure you have access to the remote machine or VM on which the full node is being set up.

  > *Refer to https://github.com/maticnetwork/node-ansible#setup for more details.*

- Clone the https://github.com/maticnetwork/node-ansible repository.
- Navigate into the node-ansible folder: `cd node-ansible`
- Edit the `inventory.yml` file and insert your IP(s) in the `sentry->hosts` section.

  > *Refer to https://github.com/maticnetwork/node-ansible#inventory for more details.*

- Check if the remote machine is reachable by running: `ansible sentry -m ping`
- To test if the correct machine is configured, run the following command:

  ```
  # Mainnet:
  ansible-playbook playbooks/network.yml --extra-var="bor_version=v0.4.0 heimdall_version=v0.3.4 network=mainnet
  node_type=sentry" --list-hosts

  # Testnet:
  ansible-playbook playbooks/network.yml --extra-var="bor_version=v0.4.0 heimdall_version=v0.3.4 network=mumbai
  node_type=sentry" --list-hosts
  ```

  <img src={useBaseUrl("img/network/full-node-mumbai.png")} />

- Next, set up the full node with this command:

  ```
  # Mainnet:
  ansible-playbook playbooks/network.yml --extra-var="bor_version=v0.4.0 heimdall_version=v0.3.4 network=mainnet
  node_type=sentry"

  # Testnet:
  ansible-playbook playbooks/network.yml --extra-var="bor_version=v0.4.0 heimdall_version=v0.3.4 network=mumbai
  node_type=sentry"
  ```

- In case you run into any issues, delete and clean the whole setup using:

  ```
  ansible-playbook playbooks/clean.yml
  ```

- Once you initiate the Ansible playbook, log in to the remote machine.
- Please **ensure that the value of seeds and bootnodes mentioned below is the same value as mentioned in Heimdall and Bor `config.toml` files**. If not, change the values accordingly.
  - Heimdall seed nodes:

    ```
    moniker=<enter unique identifier>

    # Mainnet:
    seeds="2a53a15ffc70ad41b6876ecbe05c50a66af01e20@3.211.248.31:26656,6f829065789e5b156cbbf076f9d133b4d772584 7@3.212.183

    # Testnet:
    seeds="e72c0466a02ea43b2198bd3a9454b87a3ef0d77e@54.147.31.250:26656,b302d1ddb21102e794b524d05152a7834da05bd1@34.226.1
    ```

  - Bootnodes:

```
# Mainnet:
bootnode
["enode://0cb82b395094ee4a2915e9714894627de9ed8498fb881cec6db7c65e8b9a5bd7f2f25cc84e71e89d0947e51c76e85d0847de848c778

# Testnet:
bootnodes
["enode://320553cda00dfc003f499a3ce9598029f364fbb3ed1222fdc20a94d97dcc4d8ba0cd0bfa996579dcc6d17a534741fb0a5da303a9057
```

- To check if Heimdall is synced
  - On the remote machine/VM, run `curl localhost:26657/status`
  - In the output, `catching_up` value should be `false`
- Once Heimdall is synced, run
  - `sudo service bor start`

You have successfully set up a full node with Ansible.

:::note

If Bor presents an error of permission to data, run this command to make the Bor user the owner of the Bor files:

```
sudo chown bor /var/lib/bor
```

:::

Logs

Logs can be managed by the `journalctl` linux tool. Here is a tutorial for advanced usage: [How To Use Journalctl to View and Manipulate Systemd Logs](#).

Check Heimdall node logs

```
journalctl -u heimdalld.service -f
```

Check Bor Rest-server logs

```
journalctl -u bor.service -f
```

Ports and Firewall Setup

Open ports 22, 26656 and 30303 to world (0.0.0.0/0) on sentry node firewall.

You can use VPN to restrict access for port 22 as per your requirement and security guidelines.

Overview

- Prepare the Full Node machine.
- Install Heimdall and Bor packages on the Full Node machine.
- Configure the Full node.
- Start the Full node.
- Check node health with the community.

:::note You have to follow the exact outlined sequence of actions, otherwise you will run into issues. :::

Install packages

Prerequisites

- One machine is needed.
- Bash is installed on the machine.

Heimdall

- Install the default latest version of sentry for Mainnet:

  ```
  curl -L https://raw.githubusercontent.com/maticnetwork/install/main/heimdall.sh | bash
  ```

  or install a specific version, node type (`sentry` or `validator`), and network (`mainnet` or `mumbai`). All release versions can be found on [Heimdall GitHub repository](#).

  ```
  curl -L https://raw.githubusercontent.com/maticnetwork/install/main/heimdall.sh | bash -s -- <version> <network>
  <node_type>
  # Example:
  # curl -L https://raw.githubusercontent.com/maticnetwork/install/main/heimdall.sh | bash -s -- v0.3.4 mainnet sentry
  ```

Bor

- Install the default latest version of sentry for Mainnet:

```
curl -L https://raw.githubusercontent.com/maticnetwork/install/main/bor.sh | bash
```

or install a specific version, node type (`sentry` or `validator`), and network (`mainnet` or `mumbai`). All release versions could be found on Bor Github repository.

```
curl -L https://raw.githubusercontent.com/maticnetwork/install/main/bor.sh | bash -s -- <version> <network> <node_type>

# Example:
# curl -L https://raw.githubusercontent.com/maticnetwork/install/main/bor.sh | bash -s -- v0.4.0


mainnet sentry
```

Configuration

In this section, we will go through steps to initialize and customize configurations nodes.

:::caution

Bor and Heimdall 0.3.0 use standardized paths for configuration files and chain data. If you have existing config files and chain data on your node, please skip the Configure Heimdall section below and jump directly to **Migration section** to learn about migrating configs and data to standardized file locations.

:::

Configure Heimdall

- Initialize Heimdall configs

```
# For mainnet
sudo -u heimdall heimdalld init --chain=mainnet --home /var/lib/heimdall

# For testnet
sudo -u heimdall heimdalld init --chain=mumbai --home /var/lib/heimdall
```

- You will need to add a few details in the `config.toml` file. To open the `config.toml` file run the following command `vi /var/lib/heimdall/config/config.toml`
  - Now in the config file you will have to change `Moniker`

    ```
    moniker=<enter unique identifier> For example, moniker=my-sentry-node
    ```

  - Change the value of **Prometheus** to `true`
  - Set the `max_open_connections` value to `100`

  Make sure you keep the proper formatting when you make the changes above.

Configure service files for bor and heimdall

After successfully installing Bor and Heimdall through packages, their service file could be found under `/lib/systemd/system` , and Bor's config file could be found under `/var/lib/bor/config.toml` . You will need to check and modify these files accordingly.

- Make sure the chain is set correctly in `/lib/systemd/system/heimdalld.service` file. Open the file with following command `sudo vi /lib/systemd/system/heimdalld.service`
  - In the service file, set `--chain` to `mainnet` or `mumbai` accordingly

  Save the changes in `/lib/systemd/system/heimdalld.service` .

- Make sure the chain is set correctly in `/var/lib/bor/config.toml` file. Open the file with following command `sudo vi /var/lib/bor/config.toml`
  - In the config file, set `chain` to `mainnet` or `mumbai` accordingly.
  - To enable Archive mode you can optionally enable the following flags:

    ```
    gcmode "archive"

    [jsonrpc]
      [jsonrpc.ws]
        enabled = true
        port = 8546
        corsdomain = ["*"]
    ```

  Save the changes in `/var/lib/bor/config.toml` .

Start services

Reloading service files to make sure all changes to service files are loaded correctly.

```
sudo systemctl daemon-reload
```

Start Heimdall, Heimdall rest server, and Heimdall bridge.

```
sudo service heimdalld start
```

You can also check Heimdall logs with command

```
journalctl -u heimdalld.service -f
```

Now you need to make sure that **Heimdall is synced** completely and only then Start Bor. If you start Bor without Heimdall syncing completely, you will run into issues frequently.

- To check if Heimdall is synced
  - On the remote machine/VM, run `curl localhost:26657/status`
  - In the output, `catching_up` value should be `false`

Now once Heimdall is synced, run

```
sudo service bor start
```

You can check Bor logs via command

```
journalctl -u bor.service -f
```

In this document, we will describe how to deploy Polygon nodes into a Virtual Machine instance on the Google Cloud Platform (GCP).

Before we start, please check the minimum and recommended hardware requirements.

It is recommended to use any modern Debian or Linux Ubuntu OS with long-term support, i.e. Debian 11, Ubuntu 20.04. We'll focus on Ubuntu 20.04 in this guide.

Deploy VM Instance

You may use any of the following ways to create an instance in Google Cloud:

1. Google Cloud CLI, local or Cloud Shell
2. Web Console

We'll only cover the first case in this manual. Let's start with deployment using Google Cloud CLI.

1. Follow "Before you begin" section to install and configure gcloud command-line tool. Pay attention to default region and zone, choose ones closer to you or your customers. You may use gcping.com to measure latency to choose the closest location.

2. Adjust the following command variables using your favorite editor prior executing, when required

   - `POLYGON_NETWORK` - choose `mainnet` or `mumbai` testnet network to run
   - `POLYGON_NODETYPE` - choose `archive` , `fullnode` node type to run
   - `POLYGON_BOOTSTRAP_MODE` - choose bootstrap mode `snapshot` or `from_scratch`
   - `POLYGON_RPC_PORT` - choose JSON RPC bor node port to listen on, the default value is what used on VM instance creation and in firewall rules
   - `EXTRA_VAR` - choose Bor and Heimdall branches, use `network_version=mainnet-v1` with `mainnet` network and `network_version=testnet-v4` with `mumbai` network
   - `INSTANCE_NAME` - the name of a VM instance with Polygon we are going to create
   - `INSTANCE_TYPE` - GCP machine type, default value is recommended, You may change it later if required
   - `BOR_EXT_DISK_SIZE` - additional disk size in GB to use with Bor, default value with `fullnode` is recommended, You may expand it later if required. You'll need 8192GB+ with `archive` node though
   - `HEIMDALL_EXT_DISK_SIZE` - additional disk size in GB to use with Heimdall, default value is recommended
   - `DISK_TYPE` - GCP disk type, SSD is highly recommended. You may need to increase the total SSD GB quota in the region you are spinning up the node.

3. Use the following command to create an instance with correct hardware and software requirements. In the example below, we deploy Polygon `mainnet` from `snapshot` in the `fullnode` mode:

```
   export POLYGON_NETWORK=mainnet
   export POLYGON_NODETYPE=fullnode
   export POLYGON_BOOTSTRAP_MODE=snapshot
   export POLYGON_RPC_PORT=8747
   export GCP_NETWORK_TAG=polygon
   export EXTRA_VAR=(bor_branch=v0.4.0 heimdall_branch=v0.3.4  network_version=mainnet-v1 node_type=sentry/sentry
heimdall_network=${POLYGON_NETWORK})
   gcloud compute firewall-rules create "polygon-p2p" --allow=tcp:26656,tcp:30303,udp:30303 --description="polygon p2p" -
-target-tags=${GCP_NETWORK_TAG}
   gcloud compute firewall-rules create "polygon-rpc" --allow=tcp:${POLYGON_RPC_PORT} --description="polygon rpc" --
target-tags=${GCP_NETWORK_TAG}
   export INSTANCE_NAME=polygon-0
   export INSTANCE_TYPE=e2-standard-8
```

```
    export BOR_EXT_DISK_SIZE=1024
    export HEIMDALL_EXT_DISK_SIZE=500
    export DISK_TYPE=pd-ssd
    gcloud compute instances create ${INSTANCE_NAME} \
    --image-project=ubuntu-os-cloud \
    --image-family=ubuntu-2004-lts \
    --boot-disk-size=20 \
    --boot-disk-type=${DISK_TYPE} \
    --machine-type=${INSTANCE_TYPE} \
    --create-disk=name=${INSTANCE_NAME}-bor,size=${BOR_EXT_DISK_SIZE},type=${DISK_TYPE},auto-delete=no \
    --create-disk=name=${INSTANCE_NAME}-heimdall,size=${HEIMDALL_EXT_DISK_SIZE},type=${DISK_TYPE},auto-delete=no \
    --tags=${GCP_NETWORK_TAG} \
    --metadata=user-data='
    #cloud-config

    bootcmd:
    - screen -dmS polygon su -l -c bash -c "curl -L https://raw.githubusercontent.com/maticnetwork/node-
ansible/master/install-gcp.sh | bash -s -- -n '${POLYGON_NETWORK}' -m '${POLYGON_NODETYPE}' -s
'${POLYGON_BOOTSTRAP_MODE}' -p '${POLYGON_RPC_PORT}' -e \"'${EXTRA_VAR}'\"; bash"'
```

The instance should be created and live in a couple of minutes.

Login to VM

It will take a couple of minutes to install all the required software and a couple of hours to download a snapshot when chosen.

- You should see working `bor` and `heimdalld` processes filling up additional drives. You may run the following commands to check it.

```
gcloud compute ssh ${INSTANCE_NAME}
# inside the connected session
sudo su -

ps uax|egrep "bor|heimdalld"
df -l -h
```

- You may use the following command to watch the installation progress, it's really handy in case of `snapshot` bootstrap:

```
# inside the connected session
screen -dr
```

  Use `Control+a d` key combination to disconnect from progress review.

- You may use the following commands to get Bor and Heimdall logs:

```
# inside the connected session
journalctl -fu bor
journalctl -fu heimdalld
```

:::note

Blockchain data is saved onto additional drives which are kept by default on VM instance removal. You need to remove additional disks manually if you don't need this data anymore.

:::

At the end, you will get an instance as shown in the below diagram.

<img src={useBaseUrl("img/mainnet/polygon-instance.svg")} />
--- id: archive-node title: How to Setup an Archive PoS Node sidebar_label: Run an Archive Node description: Using binaries or ansible to set up an archive node. keywords: - erigon - archive - node - binary - polygon - docs image: https://wiki.polygon.technology/img/polygon-wiki.png ---

import useBaseUrl from '@docusaurus/useBaseUrl'; import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem';

System Requirements

- 16-core CPU
- 64 GB RAM
- Basically io1 or above with at least 20k+ iops and RAID-0 based disk structure

Configuration

<Tabs defaultValue="binaries" values={[{ label: 'Binaries', value: 'binaries', }, { label: 'Ansible', value: 'ansible', }, { label: 'Erigon Client', value: 'erigon', },] }>

To setup the Archive Node, you need to follow the same process for deploying a full node using binaries.

However, it requires a minor configuration change. You should include the following parameter in the `start.sh` file:

```
--gcmode 'archive'
```

To setup an archive node using Ansible, you need to follow the same process for deploying a full node with Ansible.

However, it requires a minor configuration change. You should include the following parameter in the `start.sh` file:

```
--gcmode 'archive'
```

System Requirements

- For an Archive node of Polygon Mainnet: 8TB
- For an Archive node of Mumbai Testnet: 1TB
- SSD or NVMe. Bear in mind that SSD performance deteriorates when close to capacity
- RAM: >= 32GB, 64-bit architecture
- Golang version >= 1.18, GCC 10+

:::note HDD not recommended

On HDDs, Erigon will always stay N blocks behind the chain tip, but will not fall behind.

:::

Install Erigon Client

Run the following commands to install Erigon:

```
git clone --recurse-submodules -j8 https://github.com/maticnetwork/erigon.git
cd erigon
git checkout v0.0.8
make erigon
```

This should create the binary at `./build/bin/erigon`

Use the tag `v0.0.8` on our forked repo to have a stable version.

Start Erigon Client

To start Erigon, run:

```
erigon --chain=mumbai
```

- Use `chain=mumbai` for Mumbai testnet
- Use `chain=bor-mainnet` for Polygon Mainnet

Configure Erigon Client

- If you want to store Erigon files in a non-default location, use `-datadir`

  ```
  erigon --chain=mumbai --datadir=<your_data_dir>
  ```

- If you are not using local **heimdall**, use `-bor.heimdall=<your heimdall url>`. By default, it will try to connect to `localhost:1317`.

  ```
  erigon --chain=mumbai --bor.heimdall=<your heimdall url> --datadir=<your_data_dir>
  ```

  - If you want to connect to Polygon Mumbai Testnet use: https://heimdall-api-testnet.polygon.technology
  - For Polygon Mainnet: https://heimdall-api.polygon.technology

Note: Remote heimdall is not recommended for production use, Only use it for testing purposes.

Tips for Faster Sync

- Use the machine with high IOPS and RAM for the faster initial sync
- Memory optimized nodes are recommended for faster sync, For example, AWS EC2 `r5` or `r6` series instances.

--- id: bloxroute title: BloXroute Instructions sidebar: BloXroute Instructions description: Connect your Polygon network nodes to BloXroute gateways using BDN. keywords: - docs - matic - polygon - bloXroute - BDN image: https://wiki.polygon.technology/img/polygon-logo.png ---

As validators, we need to keep track of block propagation time, given that we offer a 2 sec block time. We are working with bloXroute to leverage their infrastructure (Blockchain Distribution Network or BDN), which would help lower the block propagation time. For this, we need your support in connecting your Polygon network nodes to bloXroute's gateways.

:::info To get your bloXroute endpoints, please ping the **Validator Support team** on Discord. :::

You will be able to add this gateway as a static (trusted) peer/node on your sentry node. If you are having any apprehensions about divulging the enode and region of your sentry node, you can rest assured that this information will be only shared with the bloXroute team and will be safe with them.

Connecting to the BDN

Step 1: Create an account at the **bloXroute user portal** and subscribe to the Introductory Plan (free of charge).

Step 2: Contact the Validator Support Team for the bloXroute endpoint.

Step 3: Once bloXroute team creates a trusted peer specifically for your node, you will receive the hosted gateway information (enode public key). To add this enode as a trusted peer in your sentry node, follow these steps:

Add the following content using `vim ~ /var/lib/bor/config.toml` :

```
[Node.P2P]
TrustedNodes = ["enode://<node_id_of_gateway_provided_by_bloxroute>"]

[Node.P2P]
StaticNodes = ["enode://static_node_enode1@ip:port", "enode://static_node_enode2@ip:port", ... ]
TrustedNodes = ["enode://<enode_id_of_gateway_provided_by_bloxroute>"]
```

Now restart Bor:

```
sudo service bor restart
```

Step 4: Inform the Polygon team on the Discord channel **#mainnet-validator-queries** once you are done adding the trusted peer. bloXroute will activate the hosted gateway and confirm the success of the connection within the next 24 hours.

---

id: change-signer-address title: Change Your Signer Address description: Change the signer address of your validator keywords: - docs - matic - polygon - signer address - change - validator slug: change-signer-address image: https://wiki.polygon.technology/img/polygon-logo.png

import useBaseUrl from '@docusaurus/useBaseUrl';

For information on what a signer address is, see Key Management.

Prerequisites

Make sure your new validator node is fully synced and is running with the new signer address.

Change the Signer Address

This guide refers to your current validator node as Node 1 and your new validator node as Node 2.

1. Log in to the staking dasboard with the Node 1 address.
2. On your profile, click **Edit Profile**.
3. In the **Signer's address** field, provide the Node 2 address.
4. In the **Signer's public key** field, provide the Node 2 public key.

   To get the public key, run the following command on the validator node:

   ```
   heimdalld show-account
   ```

Clicking Save will save your new details for your node. This essentially means that Node 1 will be your address that controls the stake, where the rewards will be sent to, etc. And Node 2 will now be performing activities like signing blocks, signing checkpoints, etc.

id: validator-index title: Validator Index description: A collection of instructions on how to run and operate validator nodes on the Polygon Network keywords: - docs - polygon - validate - validator - maintain - architecture - Validator Index slug: validator-index image: https://wiki.polygon.technology/img/polygon-logo.png

:::tip Stay in the know

Keep up with the latest node and validator updates from the Polygon team and the community by subscribing to Polygon notifications.

:::

Validators are the key actor in maintaining the Polygon network. Validators run a full node and secure the network by staking MATIC to produce blocks, validate and participate in PoS consensus.

:::info

There is limited space for accepting new validators. New validators can only join the active set when a currently active validator unbonds.

A new auction process for validator replacement will be rolled out.

:::

Overview

Polygon consists of the three following layers:

- Ethereum layer — a set of contracts on the Ethereum mainnet.
- Heimdall layer — a set of proof-of-stake Heimdall nodes running in parallel to the Ethereum mainnet, monitoring the set of staking contracts deployed on the Ethereum mainnet, and committing the Polygon Network checkpoints to the Ethereum mainnet. Heimdall is based on Tendermint.
- Bor layer — a set of block-producing Bor nodes shuffled by Heimdall nodes. Bor is based on Go Ethereum.

To be a validator on the Polygon Network, you must run:

- Sentry node — a separate machine running a Heimdall node and a Bor node. A sentry node is open to all nodes on the Polygon Network.
- Validator node — a separate machine running a Heimdall node and a Bor node. A validator node is only open to its sentry node and closed to the rest of the network.
- Stake the MATIC tokens in the staking contracts deployed on the Ethereum mainnet.

Components

Heimdall

Heimdall does the following:

- Monitors the staking contracts on the Ethereum mainnet.
- Verifies all state transitions on the Bor chain.
- Commits the Bor chain state checkpoints to the Ethereum mainnet.

Heimdall is based on Tendermint.

:::info See Also

- GitHub repository: Heimdall
- GitHub repository: Staking contracts
- Blog post: Heimdall and Bor

:::

Bor

Bor does the following:

- Produces blocks on the Polygon Network.

Bor is the Block producer node and layer for the Polygon Network. It is based on Go Ethereum. Blocks produced on Bor are validated by Heimdall nodes.

:::info See Also

- GitHub repository: Bor
- Blog post: Heimdall and Bor

:::

This section guides you through the following topics:

- Validator responsibilities
- Joining the network as a validator:
  - Start and run the nodes with Ansible
  - Start and run the nodes with binaries
  - Stake as a validator
- Maintaining your validator nodes:
  - Change the signer address
  - Change the commission

Community assistance:

- Discord
- Forum

---

id: reporting-issues title: Reporting Issues description: Report issues on attacks and bugs on the Polygon Network. keywords: - docs - polygon - matic - issues - report - bugs - attacks slug: reporting-issues image: https://wiki.polygon.technology/img/polygon-logo.png

Where to report a bug

For any bugs or attacks that are discovered, you need to report them to the Immunefi bug bountry program. **Performing an attack and not providing submission of your proof will result in disqualification of your attempt.**

You need to make sure that you add all relevant details, such as an email address and Discord ID in order to maintain a rapport of communication in case it is required. You also need to provide as many details as possible so that the Polygon team can appropriately evaluate your submission.

What happens after submitting a report

Upon reporting an issue, the Polygon team will review and update / comment on the status of the issue. Upon evaluation, the Polygon team will report the outcome of the submission. The Severity will also be tagged as per the evaluation.

Contact us for further questions

You can always connect with the community leaders, Anurag & Parvez, via email or tag the validator-support-team on Discord:

Via E-mail

- Anurag — anurag@polygon.technology
- Parvez — parvez.shaikh@polygon.technology

Via Discord

- Check out the @validator-support-team channel on our Discord server.

---

id: how-to title: Additional Prerequisites sidebar_label: Additional Prerequisites description: Polygon validator tutorials keywords: - docs - polygon - how to - knowledge - faq - node - validator slug: how-to image: https://wiki.polygon.technology/img/polygon-logo.png

import useBaseUrl from '@docusaurus/useBaseUrl';

Before setting up the Validator node

:::caution

We recommend running the Validator node with Sentry. Otherwise, you may encounter security concerns and issues with your Validator node.

:::

Downloading the Snapshot

It is recommended that you keep your snapshots handy before setting up the Validator node. Link to download and extract the snapshots: https://snapshots.polygon.technology/

RPC Endpoint for Node Setup

Validator nodes require an Ethereum-based RPC endpoint. You may use your own Ethereum node or utilize external infrastructure providers in our list. In order to see the list, click on the dropdown menu labeled **Infra Providers** in the navigation bar.

Open necessary ports

- **Port 26656** → Heimdall service will connect your node to another node's Heimdall service using this port.
- **Port 30303** → Bor service will connect your node to another node's Bor service using this port.
- **Port 22** → for the Validator to be able to SSH from wherever they are.

Install RabbitMQ

It is recommended to install the **RabbitMQ** service before setting up your Validator node. Please utilize the below commands to set up RabbitMQ (if not already installed):

```
sudo apt-get update
sudo apt install build-essential
sudo apt install erlang
wget https://github.com/rabbitmq/rabbitmq-server/releases/download/v3.10.8/rabbitmq-server_3.10.8-1_all.deb
sudo dpkg -i rabbitmq-server_3.10.8-1_all.deb
```

Mandatory Checklist for Validator Setup

1. Please follow the below checklist in order to set up your Validator node using Binaries, Ansible or Packages.

Checklist	Binaries	Ansible	Packages
Machines Required	2 Machines - **Sentry** & **Validator**	3 Machines - **Local Machine**, **Sentry** and **Validator**	2 Machines - **Sentry** & **Validator**
Install Go Packages	Yes	No	No
Install Python	No	Yes (only on the **Local Machine** where the **Ansible Playbook** runs)	No
Install Ansible	No	Yes (only on one machine)	No
Install Bash	No	No	Yes
Run Build Essential	Yes	No	No

Node Setup	Using Binaries	Using Ansible	Using Packages

2. Once your Sentry and Validator nodes are synced and running, head over to our Discord server and ask the community to health-check your nodes. You may check the logs by using the following commands:
   - **Heimdall logs →** `journalctl -u heimdalld.service -f`
   - **Bor logs →** `journalctl -u bor.service -f`
3. It is highly recommended to maintain a backup of the key files on your local machine. These might be needed in the situation of migration or outage. You may access the files using the commands below:

```
vi ~/.heimdalld/config/priv_validator_key.json
vi ~/.bor/keystore/UTC--XXXXX
vi ~/.bor/password.txt
vi /etc/matic/metadata
```

4. Ensure that Bor and Heimdall are on their right versions. Commands to verify the versions are provided below:

```
heimdalld version
bor version
```

5. It is recommended to always maintain 2 sentries to maximize your node uptime.
6. Constantly keep checking the peer count on Heimdall and Bor service using the following commands:

```
# Heimdall
curl localhost:26657/net_info? | jq .result.n_peers

# Bor
bor attach .bor/data/bor.ipcadmin.peers.length
```

:::info

For Validators, **the output should be only one peer, which has to be a Sentry**. Validators should connect only to the Sentry and not with external peers.

Sentry can connect with multiple peers.

:::

1. How to: New node setup

:::note

Below are some additional information to the official documentation which might help while setting up a new node.

:::

This document gives some additional details for setting up a new node based on Full Node Deployment guide.

- VPN setup is required before proceeding with any of these steps can be done by referring Bastillion user guide.
- If you are using Macbook, it has native Python 2.7 available which **needs to be replaced by with anything Python 3.x**.
- Make sure if pip3 packages are installed correctly

  If the above 3 steps are not done correctly then even though Ansible is installed, it can't recognize the Ansible packages. You might face something like the below screenshot:

  <img src={useBaseUrl("img/knowledge-base/node-setup-1.png")} width="100%" height="100%"/>
- Also make sure there are no Go packages and any previous setup of Bor or Heimdall.

  You can use the below commands to check whether these packages are existing or not;

  1. Go version
  2. Heimdall version
  3. Bor version

  If anything of that sort, please run the below command to delete and clean the whole setup;

```
ansible-playbook -l sentry playbooks/clean.yml
```

  Otherwise there will be error like below image whenever you try to run `ansible sentry -m ping`

  <img src={useBaseUrl("img/knowledge-base/node-setup-2.png")} width="100%" height="100%"/>
- Example of **inventory.yml**

  <img src={useBaseUrl("img/knowledge-base/node-setup-3.png")} width="100%" height="100%"/>
- Sentry Host IP and Validator Host IP have to be the same and colons should be provided at the end of the lines including IPs
- Before connecting the remote machine using the below command, you must be added to the remote machine and it will provided by DevOps team

```
ssh -i <downloaded_key_file.key> <remote_user>@<ip/host>
```

- Once they have confirmed the server access, you should be able to SSH to the remote machine.
- You may face this error if there is any issue with Heimdall's config

  <img src={useBaseUrl("img/knowledge-base/node-setup-4.png")} width="100%" height="100%"/>

  This can be fixed by following steps:

  - Run the below commands(inside the 'node-ansible' folder):

    ```
    git checkout fixing_symlinks_on_clean
    git pull https://github.com/maticnetwork/node-ansible/tree/fixing_symlinks_on_
    ```

  - Cross check `clean.yml` on your machine with clean.yml in the github repo
  - If there are any differences, replace the one on your machine with the one in the repo
  - You should be able to run the clean script now and then also be able to run the installation script

- `moniker=enter unique identifier`
  - the unique identifier asked as per the document (it can be anything, ex: your name)
- `eth_rpc_url =insert Infura or any full node RPC URL to Ethereum`
  - For this step, sign in to https://infura.io (signup if you already don't have an account)
  - Copy the **https** endpoint provided under **ethereum → keys → endpoints**

    <img src={useBaseUrl("img/knowledge-base/node-setup-5.png")} width="100%" height="100%"/>

    - Provide the copied **https** endpoint as `eth_rpc_url` in `/var/lib/heimdall/config/heimdall-config.toml`

2. Why do I have to keep ETH in my signer account?

ETH is required on your signer account because for submitting checkpoints to Ethereum, all transactions require ETH to be used as Gas. Hence, ETH is required in your Signer Account.

3. For a Polygon Validator, do I need to setup a Sentry and Validator node or can I just run the Validator node only?

For a Polygon Validator, our ecosystem and architecture demands that you run a **Sentry + Validator** setup. This is to ensure that your Validator node is not exposed to the public and only your Sentry node is.

Your Sentry node gleans information / blocks from the network and then relays them to the Validator for validation.

4. How to migrate to new nodes and then cut over?

1. Provision nodes and install all software as per the instructions.
2. Download the latest Heimdall and Bor snapshots on both nodes.
3. Move the Key and Keystore files to the new validator.
4. Shut down the current validator and sentry node.
5. Start all services on sentry, then the validator.

5. How to check the Heimdall version?

Run: `heimdalld version`

6. Which Private Key should we add when we generate validator key?

The Private key to be used is your Wallet's ETH address where your Matic testnet Tokens are stored. You can complete the setup with one public-private key pair tied to the address submitted on the form.

7. Where can we find Heimdall account info location?

For Binaries: `~/.heimdalld/config folder`

```
/var/lib/heimdall/config folder
```

For Linux package:

```
/etc/heimdall/config
```

8. Which file do we add the API key?

Once you have created the API key, you need to add the API key in `heimdall-config.toml` file.

9. Which file do we add the `persistent_peers` ?

You can add the `persistent_peers` in the following file:

```
/var/lib/heimdall/config/config.toml
```

10. How to stop Heimdall and Bor services?

For Linux packages:

Stop Heimdall: `sudo service heimdalld stop`

Stop Bor: `sudo service bor stop` or

1. `ps -aux | grep bor` . Get the PID for Bor and then run the following command.
2. `sudo kill -9 PID`

For Binaries:

Stop Heimdall: `pkill heimdalld`

Stop Bridge: `pkill heimdalld-bridge`

Stop Bor: `bash stop.sh`

11. How to remove Heimdall and Bor directories?

For Linux packages:

Delete Heimdall: `sudo rm -rf /etc/heimdall/*`

Delete Bor: `sudo rm -rf /etc/bor/*`

For Binaries:

Delete Heimdall: `sudo rm -rf /var/lib/heimdall/`

Delete Bor: `sudo rm -rf /var/lib/bor`

12. How to reduce cache in Bor?

Bor supports the `--cache` parameter which can reduce the cache to avoid running out of memory.

13. How to delete the Bor DB data?

```
bor --datadir  /var/lib/bor/data removedb
cd ~/node/bor
bash setup.sh
service bor start
```

---

id: known-issues title: Known Issues & Errors description: Knowledge Base of common errors keywords: - docs - polygon - matic - issues - errors - invalid - command slug: known-issues image: https://wiki.polygon.technology/img/polygon-logo.png

import useBaseUrl from '@docusaurus/useBaseUrl';

Error: Bad block/Invalid Merkle

Description: A Bad Block or Invalid Merkle root error occurs when your Heimdall and Bor are not in sync with each other. Heimdall is the consensus layer for Polygon POS chain, which means that Heimdall directs Bor to create blocks accordingly. A Bad Block occurs when Bor moves ahead to create a block which has not been directed by Heimdall and hence there is an invalid hash been created, which causes the error, Bad Block, or Invalid Merkle root.

Solution 1: Typically a restart of the Bor service should resolve the problem. This would ensure your Bor connects with Heimdall again and start syncing and creating blocks correctly.

To restart your Bor service you can use the following command, `sudo service bor restart` .

Solution 2: If a restart does not fix your problem, then the first thing you need to check is your Heimdall and REST server. Most of the time, your Heimdall service might have stopped which has caused the Bad block issue on Bor.

Check the logs for your Heimdall first, `journalctl -u heimdalld -f` and check if everything is working correctly.

Additionally, also check your REST server logs, `journalctl -u heimdalld-rest-server -f` .

If you find that any of these services are not running correctly, then please restart the above services and let them sync. Your Bor should automatically resolve the problem.

Solution 3: If a restart of your Bor and Heimdall services don't resolve the problem, then its probably that your Bor is stuck on a block. The block number will be evident in the logs. To check your logs for Bor you can run this command, `journalctl -u bor -f` .

The Bad block would be displayed this way in your logs:

<img src={useBaseUrl("img/knowledge-base/bad_block.png")}/>

Once you know the Bad block number, you could roll back your Blockchain by a few hundred blocks and resync from a previous block. In order to do this, you will first need to convert the Block number to hexadecimal. You can use this tool for converting decimals to hexadecimals.

Once you have your Hexadecimal ready, you can run the following commands;

```
bor attach ./.bor/data/bor.ipc
> debug.setHead("0xE92570")
```

`debug.setHead()` is the function that will allow your Bor to set the tip at a particular Block height.

Once you run these commands, the output for this would be `null` . Null means good and it is intended. You can now start monitoring your logs for Bor again and see if it passes that block number.

If in any case, none of these solutions work for you, please contact the Polygon Support team immediately.

Log: Error validating checkpoint module=checkpoint startBlock

If your node throws up these logs, please check the following:

- Ensure that the **Bor node is in sync**. To check, run the following command:

  ```
  bor attach .bor/data/bor.ipc
  eth.syncing
  ```

  If the output is **false**, then the Bor node is in sync.

- Another possible cause of this issue is that **your node might be on the wrong fork**. To check, run the following command to find the current block on your node:

  ```
  bor attach .bor/data/bor.ipc
  eth_blockNumber
  ```

  Find the associated block hash for the block number:

  ```
  bor attach .bor/data/bor.ipc
  eth.getBlockByNumber("<Block Number>").hash
  ```

  Now, you can inspect the block number to identify if you are running on the right fork. One way to do this is to search for the block number on an explorer like PolygonScan.

  <img src={useBaseUrl("img/knowledge-base/block_number.png")}/>

  If the hashes match, then the node is on the right fork.

Log: Error dialing seed

Please check whether your Heimdall node is configured with the latest seeds as listed on the node setup documents.

If you're still encountering the error after either updating to the latest seeds or confirming that you are using the right seeds, you may need to clear the `addrbook.json` file. To do this, follow the steps below.

1. Open the `config.toml` file in your terminal: `vi /var/lib/heimdall/config/config.toml`
2. Stop `heimdalld` service: `sudo service heimdalld stop`
3. Clear your `addrbook`

   ```
   sudo service heimdalld stop
   cp /var/lib/heimdall/config/addrbook.json /var/lib/heimdall/config/addrbook.json.bkp
   rm /var/lib/heimdall/config/addrbook.json
   ```

4. Increase `max_num_inbound_peers` and `max_num_outbound_peers` in `/var/lib/heimdall/config/config.toml` :

   ```
   max_num_inbound_peers = 300
   max_num_outbound_peers = 100
   ```

5. Start `heimdalld` service: `sudo service heimdalld start`

Log: Demoting invalidated transaction

This log is not an error. It is a process in **txpool** which rearranges the transactions and removes some of them (as per specific conditions).

It should in **no way affect the checkpointing mechanism**.

Error: Failed Sanity Checks

Description: `Addressbook` warnings can be ignored without an issue most of the time. If your node is connected to sufficient number of peers these kind of errors can be ignored. Your `pex` is just trying to re-establish it's connections with peers already present in `addrbook.json` .

Issue: Bor synchronisation is slow

Description: If Bor synchronisation is slow it could be due to either of the below reasons:

- The node is running on a fork - means at certain point the block production was done by forking on a different block and that has impacted the further block production
- The machine is not working at optimum levels and could be with insufficient resources.
  - This can be addressed by checking on:
    - IOPS
      - IOPS stands for Input/Output state of cycle
      - The rate of reading is usually higher than write speed
      - 6000 is the recommended range for IOPS
    - Processing Power
      - Processor has to be 8 or 16 core
      - RAM: 32 GB is the minimum; 64 GB is recommended
      - Block import should be more than 2 block for every second
    - Node sync rate should be at 15-20 blocks every 8 secs

Solution: As the issue is more about lack of hardware resources try upgrading it to double of the current specifications.

Node is not signing any checkpoints

PreRequisite: First, please point the bor_rpc_url(heimdall-config.toml) of the validator to any external infra providers and restart the services. This change helps to avoid missing checkpoints.

You can find a list of Infra Providers, on the navbar, that validators can make use of.

:::note

At this point in time, the node will not mine blocks. So once the issue is fixed, the changes made have to be reverted for the node to return to normal functionality.

:::

Description: First of all, your node not signing checkpoints could be for a multiple reasons.

Solution 1: First check if your Heimdall service is running correctly on your Sentry and Validator node. If the service has stopped abruptly or see any errors, try restarting your Heimdall service and see it comes back to normal.

Solution 2: Check your Bor service and see if it has halted abruptly or there are any errors on the logs. Try restarting your Bor service to resolve this issue.

Solution 3: Check if your Heimdall Bridge is running or not or if it has any errors in the logs. Try restarting the service and see if the issue resolves.

Issue: Validator Heimdall is unable to connect to Peers

Description: This typically means that your Sentry Heimdall is running into issues.

Solution:

- Check your Sentry Heimdall and see if the service is running fine.
- If the service is stopped then restarting the service on your Sentry should resolve this issue.
- Similarly, after fixing your sentry, a restart of your Heimdall service should also resolve the problem.

Error: Error while fetching mainchain receipt error

Description: These are normal logs. Do not do anything to your bridge.

Validator bor is stuck on block for a long time

Description: This means that your Bor on your Sentry is also stuck because your Validator gets information from your Sentry.

Solution:

- Please check your Bor logs on your sentry and see if everything is okay.
- Probably restart the Bor service first on your Bor and then simultaneously restart the Bor service on your Validator as well.

Error: (in Bor) "Failed to prepare header mining at block 0"

Description: This happens because of a formatting issue in your `static-nodes.json` file (/var/lib/bor/data/bor/static-nodes.json).

Solution:

- Ensure there are no space and no additional characters like < / > .
- If you have made any changes to the file then please restart your Bor service and you should see logs printing.

Error: "30303" or invalid command

Description: This is because you haven't created the bor keystore and the password file for it.

Solution:

Ensure that you follow all the steps from the guide setup.

Error: Impossible reorg, please file an issue

Description: Let these logs be. Your node should ideally not suffer because of this and the issue should be automatically resolved.

If your node is suffering because of this, please contact the support team immdiately.

Error: "Host not found" while setting up a node using Ansible

Description: This could be because your `inventory.yml` file may have some formatting issues.

Solution: Correct them with proper indentation and then try again.

Issue: "Dialling failed" in Heimdall

Description: This is related to connectivity and more specifically a port related problem.

Solution:

- Check to `curl localhost:26657/status` still shows the same block.
- Try a Heimdall Restart.
- Make sure that the connectivity to this port 26656 is open.
- Try adding additional peers in vi /var/lib/heimdall/config/config.toml
- Set `max_open_connection` parameter to 100.

Issue: Looking for Peers or Stopping Peer for error

Solution:

- open the `config.toml` file on your Sentry node.

  `/var/lib/heimdall/config/config.toml`

- And then find the parameter `external_address`. Once you find it this what you should be updating it with

  `tcp://<my_elastic_ip>:26656`

- Where `my_elastic_ip` is your Sentry's public IP

- Once you have updated this, all you need to do is restart your Heimdall service on your Sentry

  `sudo service heimdalld restart`

- Ensure that you're only doing this on your sentry only.

Follow the below steps for adding additional peers in `vi /var/lib/heimdall/config/config.toml`

- Stop heimdalld service

  ```
  sudo service heimdalld stop
  ```

- Clear your `addrbook`

  ```
  sudo service heimdalld stop
  cp /var/lib/heimdall/config/addrbook.json /var/lib/heimdall/config/addrbook.json.bkp
  rm /var/lib/heimdall/config/addrbook.json
  ```

- Increase `max_num_inbound_peers` and `max_num_outbound_peers` in `/var/lib/heimdall/config/config.toml`:

  ```
  max_num_inbound_peers = 300
  max_num_outbound_peers = 100
  ```

- Start heimdalld service:

  ```
  sudo service heimdalld start
  ```

Error: Error while fetching data from URL

Error sample:

```
module=span service=processor error="Error while fetching data from url:
[http://0.0.0.0:1317/bor/prepare-next-span?
chain_id=137&proposer=0x29f265b54a298df0c1b762f688e7e7c09d8790ea&span_id=2863&start_block=18317056]
(http://0.0.0.0:1317/bor/prepare-next-span?
chain_id=137&proposer=0x29f265b54a298df0c1b762f688e7e7c09d8790ea&span_id=2863&start_block=18317056), status: 400"
Aug 23 12:07:23 US-CA-SN01 bridge[2340]: E[2021-08-23|12:07:23.158] Unable to fetch next span details
module=span service=processor lastSpanId=2862
```

Solution:

Then the Heimdall Bridge needs a restart.

Error: no contract code at the given address

Solution

1. Get the right configs from Github and copy them to `/var/lib/heimdall/config` and

2. Please reset heimdall using `heimdalld unsafe-reset-all` .

Issue: Problems in starting Bor

Issue: Address is required as an argument.

Solution: You have to add address.

```
/etc/matic/metadata
```

Error: Failed to unlock account (0x...) No key for given address or file

Description:

This error happens in light of the fact that the way for the password.txt record is erroneous. You can follow the beneath steps to amend this.

Solution:

For Linux packages:

Kill Bor process

For linux:

1. `ps -aux | grep bor` . Get the PID for Bor and then run the following command.
2. `sudo kill -9 PID`

For Ansible:

1. Copy the bor keystore file to

   ```
   /etc/bor/dataDir/keystore
   ```

2. And password.txt to

   ```
   /etc/bor/dataDir/
   ```

3. Make sure you have added correct address in `/etc/bor/metadata`

For Binaries:

1. Copy the Bor keystore file to:

   ```
   /var/lib/bor/keystore/
   ```

2. And password.txt to

   ```
   /var/lib/bor/password.txt
   ```

Consequences of validator missing a checkpoint and points to investigate from our side

- Economics
  - Bad reputation for Validator
  - Rewards gets missed for Delegator
- Investigation
  - Ask for recent logs

Error: dpkg: error processing archive matic-heimdall-xxxxxxxxxx

Sample:

```
 "dpkg: error processing archive matic-heimdall_1.0.0_amd64.deb (--install): trying to overwrite '/heimdalld-rest-server.service', which is also in package matic-node 1.0.0"
```

Solution:

This occurs mainly because of a previous installation of Matic on machine. To resolve you can run: `sudo dpkg -r matic-node`

Issue: Tendermint was rest without resetting application's data

Solution:

- Reset Heimdall config data and try running the installation again;

  ```
  $ heimdalld unsafe-reset-all
  ```

```
$ rm -rf $HEIMDALLDIR/bridge
```

Issue: Bor crashed

Solution:

- Try upgrading to double the amount of RAM
- For example, their current RAM capacity is 16GB, it can be upgraded to 32GB

Error: err="insufficient funds for gas * price + value"

Description:

These logs throw up when there is no enough ETH in your signer wallet.

Solution: It is recommended to have 1 ETH in your signer wallet but can keep .5 to .75 in case you check it often enough.

Heimdall: No staking sequence exists: %s %s module=staking

If the following logs are found on a large frequency on Heimdall then this issue is related to the bridge service resetting the bridge directory fixes this issue.

```
sudo service rabbitmq-server stop
mv /var/lib/rabbitmq/mnesia /var/lib/rabbitmq/mnesia-old
sudo service rabbitmq-server start
```

Retrying again in 5 seconds to fetch data from Heimdall path=bor/span/1

These logs in Bor mean that it cannot connect to Heimdall.

The Heimdall doesn't look in sync and hence it won't have data on all the things that Bor would require.

So the recommended procedure would be to clear the historical data of both Heimdall and Bor and resync from the snapshot.

Ensure the following is fine:

1. Heimdall logs are normal or is it throwing up any errors?
2. Ensure Heimdall is fully synced by running: `curl localhost:26657/status`
3. Also ensure whether Heimdall is connected with the other peers.

```
curl localhost:26657/net_info? | jq .result.n_peers
```

If there aren't any peers, check whether the **seeds or persistent peers are rightly set on Heimdall** and **ensure Port 26656 is all open**.

Reset Heimdall

```
sudo service heimdalld stop
heimdalld unsafe-reset-all
```

Sync Heimdall from Snapshot

```
wget -c <Snapshot URL>
tar -xzvf <snapshot file> -C <HEIMDALL_DATA_DIRECTORY>
```

etherbase missing: etherbase must be explicitly specified

To fix this issue, the signer address that is used to mine must be added inside `miner.etherbase` section in the `config.toml` file.

Steps to Prune the node

Please use the below steps:

1. Check your Bor data size before pruning

```
du -sh /usr/bin/bor
```

2. Stop Bor

```
sudo service bor stop
```

3. Start **tmux** to ensure that even if your SSH connection is reset, the process is running on the remote machine tmux.
4. Start pruning

```
sudo bor snapshot prune-state --datadir  /usr/bin/bor
```

The default --datadir is `/usr/bin/bor` .

5. Once the pruning is completed, you will see success logs and details. Then start Bor again:

```
sudo service bor start
```

6. Check your Bor data size after pruning:

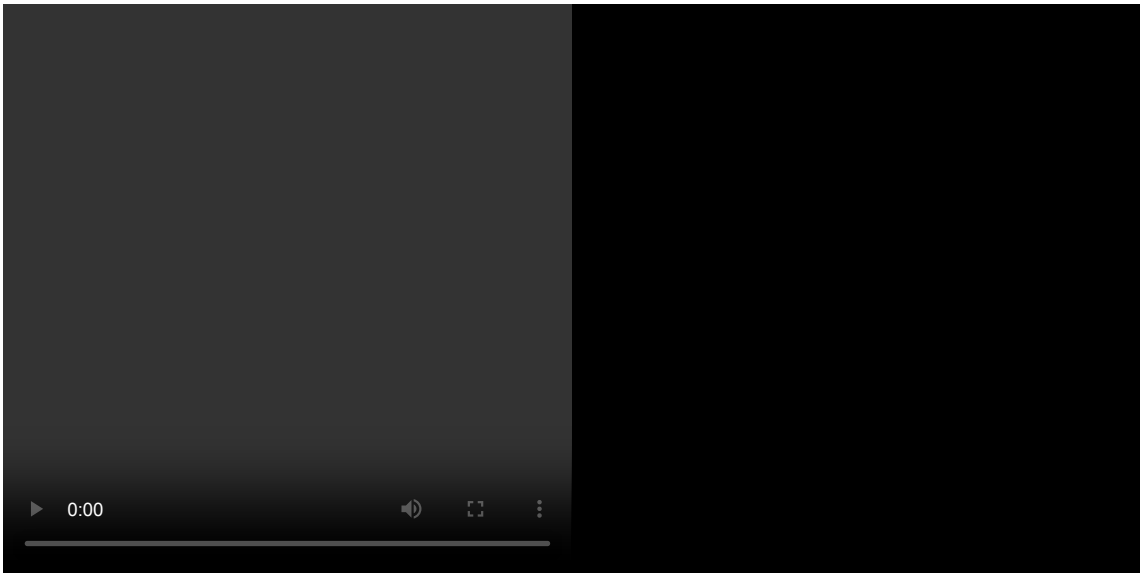
```
du -sh  /usr/bin/bor
```

---

id: move-stake title: Moving Stake description: Moving your stake on polygon network keywords: - docs - polygon - matic - stake - move stake - validator - delegator slug: move-stake image: https://wiki.polygon.technology/img/polygon-logo.png

import useBaseUrl from '@docusaurus/useBaseUrl';

Moving Stake from Foundation nodes to External Nodes



Delegators are now given an option to move their stake from the Foundation nodes to any External nodes of their choice by using the Move Stake functionality on the Staking UI

Moving Stake from the foundation node to external node is a single transaction. So there are no delays or unbonding periods durind this event.

Please note that Moving Stake is only allowed from Foundation node to External nodes. If you want to move your stake from an External node to another External node, you will have to Unbond first and then Delegate on the new external node.

Also, the Move Stake function is a temporary function developed by the Polygon team to ensure smooth transistioning of funds from the Foundation nodes to External. And will only stay active until the foundation nodes are turned off.

How to Move Stake

In order to Move stake, first you will need to login to the Staking UI using your Delegator Address.

Delegator Address : The address that you have already used for Staking on the Foundation Nodes.

Once logged in, you will see a list of Validators.

<img src={useBaseUrl("img/staking/validator-list.png")} />

Now go to your Delegator Profile by clicking on the **Show Delegator Details** button or the **My Delegator Details** option on the left.

<img src={useBaseUrl("img/staking/show-delegator-details.png")} />

Here you will find a new button called **Move Stake**.

<img src={useBaseUrl("img/staking/move-stake-button.png")} />

Clicking on that button would navigate you to a page with a list of validators that you can delegate to. You can delegate to any Validator on this list.

<img src={useBaseUrl("img/staking/move-stake-validator.png")} />

Now after choosing your validator that you want to delegate to, click on the **Delegate Here** button. Clicking on that button would open up a popup window.

<img src={useBaseUrl("img/staking/stake-funds.png")} />

Here you would see an **Amount** field which would automatically populate with entire amount for Delegation. You can also use a partial amount to delegate to a validator.

For example, if you have delegated 100 MATIC tokens to Foundation Node 1 and now you want to move your stake from the foundation node to an external node, you can delegate a partial amount to the external node of your choice, lets say 50 MATIC tokens. The rest of the 50 MATIC tokens will stay on Foundation node 1. You can then choose to either delegate the rest of the 50 tokens to another external node or the same external node.

Once you have entered the amount you can then click on **Stake Funds** button. This will then ask for confirmation on your MetaMask to sign the address.

Once you have signed the transaction your stake would have successfully moved from the Foundation node to the External node. However, you will have to wait for 12 block confirmations for it reflect on the Staking UI. If your moved funds do not show up after 12 block confirmations, try refreshing the page once to see updated stakes.

If you have any questions or any issues please submit a ticket here.

id: run-validator title: How to Run a Validator Node sidebar_label: Run a Validator Node description: Use the package to set up your validator node on the Polygon Network. keywords: - docs - polygon wiki - polygon - binary - node - validator - sentry

import useBaseUrl from '@docusaurus/useBaseUrl'; import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem';

This guide will walk you through running a Polygon validator node from packages.

For system requirements, follow the Validator Node System Requirements guide.

:::tip Snapshots Steps in this guide involve waiting for the **Heimdall** and **Bor** services to fully sync. This process takes several days to complete. Alternatively, you can use a maintained snapshot, which will reduce the sync time to a few hours. For detailed instructions, see Snapshot Instructions for Heimdall and Bor.

For snapshot download links, see Polygon Chains Snapshots. :::

▶ Port Configuration Details

<Tabs defaultValue="binaries" values={[{ label: 'Binaries', value: 'binaries', }, { label: 'Ansible', value: 'ansible', }, { label: 'Packages', value: 'package', },] }>

This guide will walk you through running a Polygon validator node from binaries.

For system requirements, follow the Validator Node System Requirements guide.

:::caution

There is limited space for accepting new validators. New validators can only join the active set when an already active validator unbonds.

:::

Prerequisites

- Two machines — one sentry and one validator.
- `build-essential` installed on both the sentry and the validator machines.

  To install:

  ```
  sudo apt-get install build-essential
  ```

- Go 1.19 installed on both the sentry and the validator machines.

  To install:

  ```
  wget https://raw.githubusercontent.com/maticnetwork/node-ansible/master/go-install.sh
  bash go-install.sh
  sudo ln -nfs ~/.go/bin/go /usr/bin/go
  ```

- RabbitMQ installed on both the sentry and the validator machines.

  Here are the commands to install RabbitMQ:

  ```
  sudo apt-get update
  sudo apt install build-essential
  sudo apt install erlang
  wget https://github.com/rabbitmq/rabbitmq-server/releases/download/v3.10.8/rabbitmq-server_3.10.8-1_all.deb
  sudo dpkg -i rabbitmq-server_3.10.8-1_all.deb
  ```

  :::tip

  Check more information about downloading and installing RabbitMQ here.

  :::

:::info Please follow the steps on bloXroute instructions to connect your nodes to the bloXroute gateways. :::

Overview

To get to a running validator node, conduct the following in this **exact sequence of steps**:

:::caution

You will run into configuration issues if these steps are performed out of sequence. It is important to keep in mind that a sentry node must always be set up before the validator node.

:::

1. Prepare two machines, one for the sentry node and one for the validator node.
2. Install the Heimdall and Bor binaries on the sentry and validator machines.
3. Set up the Heimdall and Bor service files on the sentry and validator machines.
4. Set up the Heimdall and Bor services on the sentry and validator machines.
5. Configure the sentry node.
6. Start the sentry node.
7. Configure the validator node.
8. Set the owner and signer keys.
9. Start the validator node.
10. Check node health with the community.

Installing the Binaries

Polygon node consists of 2 layers: Heimdall and Bor. Heimdall is a tendermint fork that monitors contracts in parallel with the Ethereum network. Bor is basically a Geth fork that generates blocks shuffled by Heimdall nodes.

Both binaries must be installed and run in the correct order to function properly.

Heimdall

Install the latest version of Heimdall and related services. Make sure you checkout to the correct release version. Note that the latest version, Heimdall v0.3.4, contains enhancements such as:

1. Restricting data size in state sync txs to:
   - **30Kb** when represented in **bytes**
   - **60Kb** when represented as **string**
2. Increasing the **delay time** between the contract events of different validators to ensure that the mempool doesn't get filled very quickly in case of a burst of events which can hamper the progress of the chain.

The following example shows how the data size is restricted:

```
Data - "abcd1234"
Length in string format - 8
Hex Byte representation - [171 205 18 52]
Length in byte format - 4
```

To install **Heimdall**, run the below commands:

```
curl -L https://raw.githubusercontent.com/maticnetwork/install/main/heimdall.sh | bash -s -- <heimdall_version> <network_type> <node_type>
```

heimdall_version: `valid v0.3+ release tag from https://github.com/maticnetwork/heimdall/releases` **network_type**: `mainnet` and `mumbai` **node_type**: `sentry`

That will install the `heimdalld` and `heimdallcli` binaries. Verify the installation by checking the Heimdall version on your machine:

```
heimdalld version --long
```

:::note

Before proceeding, Heimdall should be installed on both the sentry and validator machines.

:::

Installing Bor

Install the latest version of Bor, based on valid v0.3+ released version.

```
curl -L https://raw.githubusercontent.com/maticnetwork/install/main/bor.sh | bash -s -- <bor_version> <network_type> <node_type>
```

bor_version: `valid v0.3+ release tag from https://github.com/maticnetwork/bor/releases` **network_type**: `mainnet` and `mumbai` **node_type**: `sentry`

That will install the `bor` binary. Verify the installation by checking the Bor version on your machine:

```
bor version
```

:::note

Before proceeding, Bor should be installed on both the sentry and validator machines.

:::

Configuring the Sentry Node

Start by logging in to the remote sentry machine.

Configuring the Heimdall services

Open the Heimdall configuration file for editing:

```
vi ~/.heimdalld/config/config.toml
```

In `config.toml`, change the following parameters:

- `moniker` — any name. Example: `moniker = "my-sentry-node"`.
- `seeds` — the seed node addresses consisting of a node ID, an IP address, and a port.

  Use the following values:

  ```
  seeds="f4f605d60b8ffaaf15240564e58a81103510631c@159.203.9.164:26656,4fb1bc820088764a564d4f66bba1963d47d82329@44.232.55.71:26
  ```

- `pex` — set the value to `true` to enable the peer exchange. Example: `pex = true`.
- `private_peer_ids` — the node ID of Heimdall set up on the validator machine.

  To get the node ID of Heimdall on the validator machine:

  1. Log in to the validator machine.
  2. Run:

     ```
     heimdalld tendermint show-node-id
     ```

  Example: `private_peer_ids = "0ee1de0515f577700a6a4b6ad882eff1eb15f066"`.

- `prometheus` — set the value to `true` to enable the Prometheus metrics. Example: `prometheus = true`.
- `max_open_connections` — set the value to `100`. Example: `max_open_connections = 100`.

Save the changes in `config.toml`.

Configuring the Bor Service

Open the Bor configuration file for editing:

```
`vi ~/node/bor/start.sh`
```

In `start.sh`, add the boot node addresses consisting of a node ID, an IP address, and a port by adding the following line at the end of the file:

```
--bootnodes
"enode://0cb82b395094ee4a2915e9714894627de9ed8498fb881cec6db7c65e8b9a5bd7f2f25cc84e71e89d0947e51c76e85d0847de848c7782b13c0255247a6
```

Save the changes in `start.sh`.

Configuring a firewall

The sentry machine must have the following ports open to the world `0.0.0.0/0`:

- `26656` - Your Heimdall service will connect your node to other nodes Heimdall service.
- `30303` - Your Bor service will connect your node to other nodes Bor service.
- `22` - Open this port if your node is servicing validators. You will likely want to restrict what traffic can access this port as it is a sensitive port.

Starting the Sentry Node

You will first start the Heimdall service. Once the Heimdall service syncs, you will start the Bor service.

:::note

As mentioned earlier, the Heimdall service takes several days to sync from scratch fully.

Alternatively, you can use a maintained snapshot, which will reduce the sync time to a few hours. For detailed instructions, see [Snapshot Instructions for Heimdall and Bor](#).

For snapshot download links, see [Polygon Chains Snapshots](#).

:::

Starting the Heimdall service

Start the Heimdall service:

```
sudo service heimdalld start
```

Start the Heimdall rest-server:

```
sudo service heimdalld-rest-server start
```

Check the Heimdall service logs:

```
journalctl -u heimdalld.service -f
```

:::note

In the logs, you may see the following errors:

- `Stopping peer for error`
- `MConnection flush failed`
- `use of closed network connection`

These logs mean that one of the nodes on the network refused a connection to your node. Wait for your node to crawl more nodes on the network; you do not need to do anything to address these errors.

:::

Check the Heimdall rest-server logs:

```
journalctl -u heimdalld-rest-server.service -f
```

Check the sync status of Heimdall:

```
curl localhost:26657/status
```

In the output, the `catching_up` value is:

- `true` — the Heimdall service is syncing.
- `false` — the Heimdall service is fully synced.

Wait for the Heimdall service to sync fully.

Starting the Bor service

Once the Heimdall service syncs, start the Bor service.

Start the Bor service:

```
sudo service bor start
```

Check the Bor service logs:

```
journalctl -u bor.service -f
```

Configuring the Validator Node

:::note

To complete this section, you must have an RPC endpoint of your fully synced Ethereum mainnet node ready.

:::

Configuring the Heimdall service

Log in to the remote validator machine.

Open for editing `vi ~/.heimdalld/config/config.toml`.

In `config.toml`, change the following:

- `moniker` — any name. Example: `moniker = "my-validator-node"`.
- `pex` — set the value to `false` to disable the peer exchange. Example: `pex = false`.
- `private_peer_ids` — comment out the value to disable it. Example: `# private_peer_ids = ""`.

  To get the node ID of Heimdall on the sentry machine:

  1. Log in to the sentry machine.

2. Run `heimdalld tendermint show-node-id`.

Example: `persistent_peers = "sentry_machineNodeID@sentry_instance_ip:26656"`

- `prometheus` — set the value to `true` to enable the Prometheus metrics. Example: `prometheus = true`.

Save the changes in `config.toml`.

Open for editing `vi ~/.heimdalld/config/heimdall-config.toml`.

In `heimdall-config.toml`, change the following:

- `eth_rpc_url` — an RPC endpoint for a fully synced Ethereum mainnet node, i.e Infura. `eth_rpc_url =<insert Infura or any full node RPC URL to Ethereum>`

Example: `eth_rpc_url = "https://nd-123-456-789.p2pify.com/60f2a23810ba11c827d3da642802412a"`

Save the changes in `heimdall-config.toml`.

Configuring the Bor service

Open for editing `vi ~/.bor/data/bor/static-nodes.json`.

In `static-nodes.json`, change the following:

- `"<replace with enode://sentry_machine_enodeID@sentry_machine_ip:30303>"` — the node ID and IP address of Bor set up on the sentry machine.

  To get the node ID of Bor on the sentry machine:

  1. Log in to the sentry machine.
  2. Run `bootnode -nodekey ~/.bor/data/bor/nodekey -writeaddress`.

  Example:
  `"enode://a8024075291c0dd3467f5af51a05d531f9e518d6cd229336156eb6545581859e8997a80bc679fdb7a3bd7473744c57eeb3411719b973b2d6c69eff`

Save the changes in `static-nodes.json`.

Setting the Owner and Signer Key

On Polygon, it is recommended that you keep the owner and signer keys different.

- Signer — the address that signs the checkpoint transactions. The recommendation is to keep at least 1 ETH on the signer address.
- Owner — the address that does the staking transactions. The recommendation is to keep the MATIC tokens on the owner address.

Generating a Heimdall private key

You must generate a Heimdall private key only on the validator machine. Do not generate a Heimdall private key on the sentry machine.

To generate the private key, run:

```
heimdallcli generate-validatorkey ETHEREUM_PRIVATE_KEY
```

where

- ETHEREUM_PRIVATE_KEY — your Ethereum wallet's private key.

This will generate `priv_validator_key.json`. Move the generated JSON file to the Heimdall configuration directory:

```
mv ./priv_validator_key.json ~/.heimdalld/config
```

Generating a Bor keystore file

You must generate a Bor keystore file only on the validator machine. Do not generate a Bor keystore file on the sentry machine.

To generate the private key, run:

```
heimdallcli generate-keystore ETHEREUM_PRIVATE_KEY
```

where

- ETHEREUM_PRIVATE_KEY — your Ethereum wallet's private key.

When prompted, set up a password to the keystore file.

This will generate a `UTC-<time>-<address>` keystore file.

Move the generated keystore file to the Bor configuration directory:

```
mv ./UTC-<time>-<address> ~/.bor/keystore/
```

Add password.txt

Make sure to create a `password.txt` file then add the Bor keystore file password right in the `~/.bor/password.txt` file.

Add your Ethereum address

Open for editing `vi /etc/matic/metadata`.

In `metadata`, add your Ethereum address. Example: `VALIDATOR_ADDRESS=0xca67a8D767e45056DC92384b488E9Af654d78DE2`.

Save the changes in `metadata`.

Starting the Validator Node

At this point, you must have:

- The Heimdall service on the sentry machine syncs and is running.
- The Bor service on the sentry machine running.
- The Heimdall service and the Bor service on the validator machine configured.
- Your owner and signer keys configured.

Starting the Heimdall service

You will now start the Heimdall service on the validator machine. Once the Heimdall service syncs, you will start the Bor service on the validator machine.

Start the Heimdall service:

```
sudo service heimdalld start
```

Start the Heimdall rest-server:

```
sudo service heimdalld-rest-server start
```

Start the Heimdall bridge:

```
sudo service heimdalld-bridge start
```

Check the Heimdall service logs:

```
journalctl -u heimdalld.service -f
```

Check the Heimdall rest-server logs:

```
journalctl -u heimdalld-rest-server.service -f
```

Check the Heimdall bridge logs:

```
journalctl -u heimdalld-bridge.service -f
```

Check the sync status of Heimdall:

```
curl localhost:26657/status
```

In the output, the `catching_up` value is:

- `true` — the Heimdall service is syncing.
- `false` — the Heimdall service is synced.

Wait for the Heimdall service to fully sync.

Starting the Bor service

Once the Heimdall service on the validator machine syncs, start the Bor service on the validator machine.

Start the Bor service:

```
sudo service bor start
```

Check the Bor service logs:

```
journalctl -u bor.service -f
```

Seed nodes and bootnodes

- Heimdall seed nodes:

  ```
  moniker=<enter unique identifier>
  ```

```
# Mainnet:
seeds="2a53a15ffc70ad41b6876ecbe05c50a66af01e20@3.211.248.31:26656,6f829065789e5b156cbbf076f9d133b4d7725847@3.212.183.151:26

# Testnet:
seeds="e72c0466a02ea43b2198bd3a9454b87a3ef0d77e@54.147.31.250:26656,b302d1ddb21102e794b524d05152a7834da05bd1@34.226.134.117:
```

- Bootnodes:

```
# Mainnet:
bootnode
["enode://0cb82b395094ee4a2915e9714894627de9ed8498fb881cec6db7c65e8b9a5bd7f2f25cc84e71e89d0947e51c76e85d0847de848c7782b13c02

# Testnet:
bootnodes
["enode://320553cda00dfc003f499a3ce9598029f364fbb3ed1222fdc20a94d97dcc4d8ba0cd0bfa996579dcc6d17a534741fb0a5da303a90579431259
```

Health Checks with the Community

Now that your sentry and validator nodes are in sync and running, head over to Discord and ask the community to health-check your nodes.

:::note

As validators, it's mandatory to always have a check of the signer address. If the ETH balance reaches below 0.5 ETH then it should be refilled. Avoiding this will push out nodes from submitting checkpoint transactions.

:::

Next Steps: Staking

Now that you have your sentry and validator nodes are health-checked, proceed to the Staking guide to start backing the network.
This section guides you through starting and running the validator node through an Ansible playbook.

For the system requirements, see Validator Node System Requirements.

If you would like to start and run the validator node from binaries, see Run a Validator Node from Binaries.

:::caution

There is limited space for accepting new validators. New validators can only join the active set when an already active validator unbonds.

:::

Prerequisites

- Three machines — one local machine on which you will run the Ansible playbook; two remote machines — one sentry and one validator.
- On the local machine, Ansible installed.
- On the local machine, Python 3.x installed.
- On the remote machines, make sure Go is *not* installed.
- On the remote machines, your local machine's SSH public key is on the remote machines to let Ansible connect to them.
- We have Bloxroute available as a relay network. If you need a gateway to be added as your Trusted Peer please contact **@validator-support-team** in Polygon Discord > POS VALIDATORS | FULL NODE PROVIDERS | PARTNERS > bloxroute.

:::info

Please follow the steps on bloXroute instructions to connect your nodes to the bloXroute gateways.

:::

Overview

:::caution

You must follow the **exact outlined sequence of actions**, otherwise you will run into issues. For example, **a sentry node must always be set up before the validator node**.

:::

To get to a running validator node, do the following:

1. Have the three machines prepared.
2. Set up a sentry node through Ansible.
3. Set up a validator node through Ansible.
4. Configure the sentry node.
5. Start the sentry node.
6. Configure the validator node.
7. Set the owner and signer keys.
8. Start the validator node.
9. Check node health with the community.

Set up the Sentry node

On your local machine, clone the [node-ansible repository]():

```
git clone https://github.com/maticnetwork/node-ansible
```

Change to the cloned repository:

```
cd node-ansible
```

Add the IP addresses of the remote machines that will become a sentry node and a validator node to the `inventory.yml` file.

```
all:
  hosts:
  children:
    sentry:
      hosts:
        xxx.xxx.xx.xx: # <----- Add IP for sentry node
        xxx.xxx.xx.xx: # <----- Add IP for second sentry node (optional)
    validator:
      hosts:
        xxx.xxx.xx.xx: # <----- Add IP for validator node
```

Example:

```
all:
  hosts:
  children:
    sentry:
      hosts:
        188.166.216.25:
    validator:
      hosts:
        134.209.100.175:
```

Check that the remote sentry machine is reachable. On the local machine, run:

```
$ ansible sentry -m ping
```

You should get this as output:

```
xxx.xxx.xx.xx | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3"
    },
    "changed": false,
    "ping": "pong"
}
```

Do a test run of the sentry node setup:

```
ansible-playbook -l sentry playbooks/network.yml --extra-var="bor_version=v0.4.0 heimdall_version=v0.3.4
network_version=mainnet-v1 node_type=sentry/sentry heimdall_network=mainnet" --list-hosts
```

This will be the output:

```
playbook: playbooks/network.yml
  pattern: ['all']
  host (1):
    xx.xxx.x.xxx
```

Run the sentry node setup with sudo privileges:

```
ansible-playbook -l sentry playbooks/network.yml --extra-var="bor_version=v0.4.0 heimdall_version=v0.3.4
network_version=mainnet-v1 node_type=sentry/sentry heimdall_network=mainnet" --ask-become-pass
```

Once the setup is complete, you will see a message of completion on the terminal.

:::note

If you run into an issue and would like to start over, run:

```
ansible-playbook -l sentry playbooks/clean.yml
```

:::

Set up the Validator node

At this point, you have the sentry node set up.

On your local machine, you also have the Ansible playbook set up to run the validator node setup.

Check that the remote validator machine is reachable. On the local machine, run `ansible validator -m ping`.

You should get this as output:

```
xxx.xxx.xx.xx | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3"
    },
    "changed": false,
    "ping": "pong"
}
```

Do a test run of the validator node setup:

```
ansible-playbook -l validator playbooks/network.yml --extra-var="bor_version=v0.4.0 heimdall_version=v0.3.4
network_version=mainnet-v1 node_type=sentry/validator heimdall_network=mainnet" --list-hosts
```

You should get this as output:

```
playbook: playbooks/network.yml
  pattern: ['all']
  host (1):
    xx.xxx.x.xxx
```

Run the validator node setup with sudo privileges:

```
ansible-playbook -l validator playbooks/network.yml --extra-var="bor_version=v0.4.0 heimdall_version=v0.3.4
network_version=mainnet-v1 node_type=sentry/validator heimdall_network=mainnet" --ask-become-pass
```

Once the setup is complete, you will see a message of completion on the terminal.

:::note

If you run into an issue and would like to start over, run:

```
ansible-playbook -l validator playbooks/clean.yml
```

:::

Configure the Sentry node

Log into the remote sentry machine.

Configure the Heimdall Service

Open `config.toml` for editing `vi /var/lib/heimdall/config/config.toml`.

Change the following:

- `moniker` — any name. Example: `moniker = "my-full-node"`.
- `seeds` — the seed node addresses consisting of a node ID, an IP address, and a port.

  Should already have the following values:

  ```
  seeds="d3a8990f61bb3657da1664fe437d4993c4599a7e@3.211.248.31:26656,d3d7d397339c9126235dfab11bf925e269776f4f@3.212.183.151:26
  ```

- `pex` — set the value to `true` to enable the peer exchange. Example: `pex = true`.
- `private_peer_ids` — the node ID of Heimdall set up on the validator machine.

  To get the node ID of Heimdall on the validator machine:

  1. Log into the validator machine.
  2. Run `heimdalld tendermint show-node-id`.

  Example: `private_peer_ids = "0ee1de0515f577700a6a4b6ad882eff1eb15f066"`.

- `prometheus` — set the value to `true` to enable the Prometheus metrics. Example: `prometheus = true`.
- `max_open_connections` — set the value to `100`. Example: `max_open_connections = 100`.

Save the changes in `config.toml`.

Configure the Bor Service

Open for editing `vi /var/lib/bor/config.toml`.

In `config.sh`, ensure the boot node addresses consisting of a node ID, an IP address, and a port by the bootnode paramater:

```
bootnodes
"enode://0cb82b395094ee4a2915e9714894627de9ed8498fb881cec6db7c65e8b9a5bd7f2f25cc84e71e89d0947e51c76e85d0847de848c7782b13c0255247a6
```

Save the changes in `config.toml`.

Open for editing `vi /var/lib/bor/config.toml`.

In `config.toml`, ensure the `static-nodes` parameter has the following values:

- `"enode://validator_machine_enodeID@validator_machine_ip:30303"` — the node ID and IP address of Bor set up on the validator machine.

  To get the node ID of Bor on the validator machine:

  1. Log into the validator machine.
  2. Run `bor bootnode -node-key /var/lib/bor/data/bor/nodekey`, this command only works while bor is not running. If the ip address is `0.0.0.0` then replace it with external facing ip address

  Example:
  `"enode://410e359736bcd3a58181cf55d54d4e0bbd6db2939c5f548426be7d18b8fd755a0ceb730fe5cf7510c6fa6f0870e388277c5f4c717af66d53c440fe`

Save the changes in `config.toml`.

Configure firewall

The sentry machine must have the following ports open to the world `0.0.0.0/0`:

- `26656` - Your Heimdall service will connect your node to other nodes using the Heimdall service.
- `30303` - Your Bor service will connect your node to other nodes using the Bor service.
- `22` - Open this port if your node is servicing validators. You will likely want to restrict what traffic can access this port as it is a sensitive port.

:::note

However, if they use a VPN connection, they can allow incoming SSH connections only from the VPN IP address.

:::

Start the sentry node

You will first start the Heimdall service. Once the Heimdall service syncs, you will start the Bor service.

:::note

The Heimdall service takes several days to fully sync from scratch.

Alternatively, you can use a maintained snapshot, which will reduce the sync time to a few hours. For detailed instructions, see Snapshot Instructions for Heimdall and Bor.

For snapshot download links, see Polygon Chains Snapshots.

:::

Start the Heimdall service

The latest version, Heimdall v0.3.4, contains a few enhancements such as:

1. Restricting data size in state sync txs to:
   - **30Kb** when represented in **bytes**
   - **60Kb** when represented as **string**.
2. Increasing the **delay time** between the contract events of different validators to ensure that the mempool doesn't get filled very quickly in case of a burst of events which can hamper the progress of the chain.

The following example shows how the data size is restricted:

```
Data - "abcd1234"
Length in string format - 8
Hex Byte representation - [171 205 18 52]
Length in byte format - 4
```

Start the Heimdall service:

```
sudo service heimdalld start
```

From v.0.3.0 the Heimdall rest-server is incorporated into Heimdall. Versions before v.0.3.0 will need to start the heimdall rest-server. Later versions can skip this action Start the Heimdall rest-server:

```
sudo service heimdalld-rest-server start
```

Check the Heimdall service logs:

```
journalctl -u heimdalld.service -f
```

:::note

In the logs, you may see the following errors:

- `Stopping peer for error`
- `MConnection flush failed`
- `use of closed network connection`

These mean that one of the nodes on the network refused a connection to your node. You do not need to do anything with these errors. Wait for your node to crawl more nodes on the network.

:::

Check the Heimdall rest-server logs if your install is before v.0.3.0:

```
journalctl -u heimdalld-rest-server.service -f
```

Check the sync status of Heimdall:

```
curl localhost:26657/status
```

In the output, the `catching_up` value is:

- `true` — the Heimdall service is syncing.
- `false` — the Heimdall service is fully synced.

Wait for the Heimdall service to fully sync.

Start the Bor Service

Once the Heimdall service is fully synced, start the Bor service.

Start the Bor service:

```
sudo service bor start
```

Check the Bor service logs:

```
journalctl -u bor.service -f
```

Configure the validator node

:::note

To complete this section, you must have your own RPC endpoint of your own fully synced Ethereum mainnet node ready. The use of Infura and Alchemy is also sufficient and widely used among validators.

:::

Configure the Heimdall Service

Log into the remote validator machine.

Open `config.toml` for editing `vi /var/lib/heimdall/config/config.toml`.

Change the following:

- `moniker` — any name. Example: `moniker = "my-validator-node"`.
- `pex` — set the value to `false` to disable the peer exchange. Example: `pex = false`.
- `private_peer_ids` — comment out the value to disable it. Example: `# private_peer_ids = ""`.

  To get the node ID of Heimdall on the sentry machine:

  1. Login to the sentry machine.
  2. Run `heimdalld tendermint show-node-id`.

  Example: `persistent_peers = "sentry_machineNodeID@sentry_instance_ip:26656"`

- `prometheus` — set the value to `true` to enable the Prometheus metrics. Example: `prometheus = true`.

Save the changes in `config.toml`.

Open for editing `vi /var/lib/heimdall/config/heimdall-config.toml`.

In `heimdall-config.toml`, change the following:

- `eth_rpc_url` — an RPC endpoint for a fully synced Ethereum mainnet node, i.e Infura. `eth_rpc_url =<insert Infura or any full node RPC URL to Ethereum>`

Example: `eth_rpc_url = "https://nd-123-456-789.p2pify.com/60f2a23810ba11c827d3da642802412a"`

Save the changes in `heimdall-config.toml`.

Configure the Bor Service

Open for editing `vi /var/lib/bor/config.toml`.

In `config.toml`, ensure the trusted-nodes parameter has the following values:

- `"enode://sentry_machine_enodeID@sentry_machine_ip:30303"` — the node ID and IP address of Bor set up on the sentry machine.

  To get the node ID of Bor on the sentry machine:

  1. Log into the sentry machine.
  2. Run `bor bootnode -node-key /var/lib/bor/data/bor/nodekey`, this command only works while Bor is not running. If the IP address is `0.0.0.0`, replace it with external facing IP address

  Example:
  `"enode://a8024075291c0dd3467f5af51a05d531f9e518d6cd229336156eb6545581859e8997a80bc679fdb7a3bd7473744c57eeb3411719b973b2d6c69eff`

Save the changes in `config.toml` file.

Set the owner and signer key

:::note

To complete this section, you must have already created two Ethereum wallets and have the private keys available as needed. One address will be used as the `Signer` and the other address as the `Owner`. Only the `Signer` address will be used in this section at the moment.

:::

On Polygon, you should keep the owner and signer keys different.

- Signer — the address that signs the checkpoint transactions. The recommendation is to keep at least 1 ETH on the signer address.
- Owner — the address that does the staking transactions. The recommendation is to keep the MATIC tokens on the owner address.

Generate a Heimdall private key

You must generate a Heimdall private key only on the validator machine. **Do not generate a Heimdall private key on the sentry machine.**

To generate the private key, run:

```
heimdallcli generate-validatorkey ETHEREUM_PRIVATE_KEY
```

:::note

ETHEREUM_PRIVATE_KEY — your Ethereum wallet's `Signer` private key

:::

This will generate `priv_validator_key.json`. Move the generated JSON file to the Heimdall configuration directory:

```
mv ./priv_validator_key.json /var/lib/heimdall/config/
```

Generate a Bor keystore file

You must generate a Bor keystore file only on the validator machine. **Do not generate a Bor keystore file on the sentry machine.**

To generate the private key, run:

```
heimdallcli generate-keystore ETHEREUM_PRIVATE_KEY
```

:::note

ETHEREUM_PRIVATE_KEY — your Ethereum wallet's `Signer` private key.

:::

When prompted, set up a password to the keystore file.

This will generate a `UTC-<time>-<address>` keystore file.

Move the generated keystore file to the Bor configuration directory:

```
mv ./UTC-<time>-<address> /var/lib/bor/data/keystore/
```

Ensure the `keystore` parameter in `/var/lib/bor/config.toml` matches the directory `/var/lib/bor/data/keystore/`

Add `password.txt`

Make sure to create a `password.txt` file then add the Bor keystore file password right in the `/var/lib/bor/data/password.txt` file. Ensure that `password` parameter in `/var/lib/bor/config.toml` matches the location of the password file.

Add your Ethereum address

Open for editing `vi /var/lib/bor/config.toml`.

In `[accounts]` section you should have paramater `password` already defined from previous step, now add your Ethereum address to `unlock` parameter and also ensure `allow-insecure-unlock` has a value of `true`.

Example:

```
[accounts]
   allow-insecure-unlock = true
   password = "/var/lib/bor/password.txt"
   unlock = ["0xca67a8D767e45056DC92384b488E9Af654d78DE2"]
```

Save the changes in `/var/lib/bor/config.toml`.

Start the Validator node

At this point, you must have:

- The Heimdall service on the sentry machine fully synced and running.
- The Bor service on the sentry machine running.
- The Heimdall service and the Bor service on the validator machine configured.
- Your owner and signer keys configured.

Start the Heimdall Service

You will now start the Heimdall service on the validator machine. Once the Heimdall service syncs, you will start the Bor service on the validator machine.

Start the Heimdall service:

```
sudo service heimdalld start
```

Start the Heimdall rest-server (Versions before v.0.3.0):

```
sudo service heimdalld-rest-server start
```

Start the Heimdall bridge (Versions before v.0.3.0):

```
sudo service heimdalld-bridge start
```

Check the Heimdall service logs:

```
journalctl -u heimdalld.service -f
```

Check the Heimdall rest-server logs (Versions before v.0.3.0):

```
journalctl -u heimdalld-rest-server.service -f
```

Check the Heimdall bridge logs (Versions before v.0.3.0):

```
journalctl -u heimdalld-bridge.service -f
```

:::note

If your Heimdall version is after v.0.3.0 then the logs of heimdalld-rest-server and heimdalld-bridge are in the heimdalld service logs as these three were combined into one service from v.0.3.0.

:::

Check the sync status of Heimdall:

```
curl localhost:26657/status
```

In the output, the `catching_up` value is:

- `true` — the Heimdall service is syncing.
- `false` — the Heimdall service is fully synced.

Wait for the Heimdall service to fully sync.

Start the Bor Service

Once the Heimdall service on the validator machine is fully synced, start the Bor service on the validator machine.

Start the Bor service:

```
sudo service bor start
```

Check the Bor service logs:

```
journalctl -u bor.service -f
```

Seed nodes and bootnodes

- Heimdall seed nodes:

```
moniker=<enter unique identifier>

# Mainnet:
seeds="2a53a15ffc70ad41b6876ecbe05c50a66af01e20@3.211.248.31:26656,6f829065789e5b156cbbf076f9d133b4d7725847@3.212.183.151:26

# Testnet:
seeds="e72c0466a02ea43b2198bd3a9454b87a3ef0d77e@54.147.31.250:26656,b302d1ddb21102e794b524d05152a7834da05bd1@34.226.134.117:
```

- Bootnodes:

```
# Mainnet:
bootnode
["enode://0cb82b395094ee4a2915e9714894627de9ed8498fb881cec6db7c65e8b9a5bd7f2f25cc84e71e89d0947e51c76e85d0847de848c7782b13c02

# Testnet:
bootnodes
["enode://320553cda00dfc003f499a3ce9598029f364fbb3ed1222fdc20a94d97dcc4d8ba0cd0bfa996579dcc6d17a534741fb0a5da303a90579431259
```

Check node health with the community

Now that your Sentry and Validator nodes are synced and running, head over to Discord and ask the community to health-check your nodes.

:::note

As validators, it's mandatory to always have a check of the signer address. If the ETH balance reaches below 0.5 ETH then it should be refilled. Avoiding this will push out nodes from submitting checkpoint transactions.

:::

Proceed to staking

Now that you have your Sentry and Validator nodes health-checked, proceed to Staking.

Prerequisites

- Two machines — one sentry and one validator.
- Bash is installed on both the sentry and the validator machines.
- RabbitMQ installed on both the sentry and the validator machines. See Downloading and Installing RabbitMQ.

:::info

Please follow the steps on bloXroute instructions to connect your nodes to the bloXroute gateways.

:::

Overview

To get to a running validator node, conduct the following in this **exact sequence of steps**:

:::caution

You will run into configuration issues if these steps are performed out of sequence. It is important to keep in mind that a sentry node must always be set up before the validator node.

:::

1. Prepare two machines, one for the sentry node and one for the validator node.

2. Install the Heimdall and Bor binaries on the sentry and validator machines.
3. Set up the Heimdall and Bor service files on the sentry and validator machines.
4. Set up the Heimdall and Bor services on the sentry and validator machines.
5. Configure the sentry node.
6. Start the sentry node.
7. Configure the validator node.
8. Set the owner and signer keys.
9. Start the validator node.
10. Check node health with the community.

Installing package

Heimdall

- Install the default latest version of sentry for the Polygon Mainnet:

  ```
  curl -L https://raw.githubusercontent.com/maticnetwork/install/main/heimdall.sh | bash
  ```

  or install a specific version, node type (`sentry` or `validator`), and network (`mainnet` or `mumbai`). All release versions can be found on Heimdall GitHub repository.

  ```
  curl -L https://raw.githubusercontent.com/maticnetwork/install/main/heimdall.sh | bash -s -- <version> <network>
  <node_type>
  # Example:
  # curl -L https://raw.githubusercontent.com/maticnetwork/install/main/heimdall.sh | bash -s -- v0.3.4 mainnet sentry
  ```

Bor

- Install the default latest version of sentry for Mainnet:

  ```
  curl -L https://raw.githubusercontent.com/maticnetwork/install/main/bor.sh | bash
  ```

  or install a specific version, node type (`sentry` or `validator`), and network (`mainnet` or `mumbai`). All release versions could be found on Bor Github repository.

  ```
  curl -L https://raw.githubusercontent.com/maticnetwork/install/main/bor.sh | bash -s -- <version> <network> <node_type>

  # Example:
  # curl -L https://raw.githubusercontent.com/maticnetwork/install/main/bor.sh | bash -s -- v0.4.0 mainnet sentry
  ```

Check installation

- Check Heimdall installation

  ```
  heimdalld version --long
  ```

- Check Bor installation

  ```
  bor version
  ```

:::note

Before proceeding, Bor should be installed on both the sentry and validator machines.

:::

Configuration

In this section, we will go through steps to initialize and customize configurations nodes.

:::caution

Bor and Heimdall 0.3.0 uses standardized paths for configuration files and chain data. If you have existing config files and chain data on your node, please skip this section and jump directly to **Migration section** to learn about migrating configs and data to standardized file locations.

:::

Configure Heimdall

- Initialize Heimdall configs

```
# For mainnet
sudo -u heimdall heimdalld init --chain=mainnet --home /var/lib/heimdall
```

```
# For testnet
sudo -u heimdall heimdalld init --chain=mumbai --home /var/lib/heimdall
```

Open the Heimdall configuration file for editing:

```
vi /var/lib/heimdall/config/config.toml
```

In `config.toml`, change the following parameters:

- `moniker` — any name. Example: `moniker = "my-sentry-node"`.
- `seeds` — the seed node addresses consisting of a node ID, an IP address, and a port.

  Use the following values:

  ```
  seeds="f4f605d60b8ffaaf15240564e58a81103510631c@159.203.9.164:26656,4fb1bc820088764a564d4f66bba1963d47d82329@44.232.55.71:26
  ```

- `pex` — set the value to `true` to enable the peer exchange. Example: `pex = true`.
- `private_peer_ids` — the node ID of Heimdall set up on the validator machine.

  To get the node ID of Heimdall on the validator machine:

  1. Log in to the validator machine.
  2. Run:

     ```
     heimdalld tendermint show-node-id
     ```

  Example: `private_peer_ids = "0ee1de0515f577700a6a4b6ad882eff1eb15f066"`.

- `prometheus` — set the value to `true` to enable the Prometheus metrics. Example: `prometheus = true`.
- `max_open_connections` — set the value to `100`. Example: `max_open_connections = 100`.

Save the changes in `config.toml`.

Configure Bor

In `/var/lib/bor/config.toml`, add the following:

```
[p2p]
    [p2p.discovery]
        static-nodes = ["<replace with enode://validator_machine_enodeID@validator_machine_ip:30303>"]
```

To get the node ID of Bor on the validator machine:

1. Log into the validator machine.
2. Run `bor bootnode -node-key /var/lib/bor/data/bor/nodekey -dry-run`.

Example output:
`"enode://410e359736bcd3a58181cf55d54d4e0bbd6db2939c5f548426be7d18b8fd755a0ceb730fe5cf7510c6fa6f0870e388277c5f4c717af66d53c440feedffb`

Example content of static node field in `/var/lib/bor/config.toml`:

```
[p2p]
    [p2p.discovery]
        static-nodes =
["enode://410e359736bcd3a58181cf55d54d4e0bbd6db2939c5f548426be7d18b8fd755a0ceb730fe5cf7510c6fa6f0870e388277c5f4c717af66d53c440feedf
```

Save the changes in `/var/lib/bor/config.toml`.

Configuring a firewall

The sentry machine must have the following ports open to the world `0.0.0.0/0`:

- `26656` - Your Heimdall service will connect your node to other nodes Heimdall service.
- `30303` - Your Bor service will connect your node to other nodes Bor service.
- `22` - Open this port if your node is servicing validators. You will likely want to restrict what traffic can access this port as it is a sensitive port.

Configure service files for Bor and Heimdall

After successfully installing Bor and Heimdall through packages, their service file could be found under `/lib/systemd/system`, and Bor's config file could be found under `/var/lib/bor/config.toml`. You will need to check and modify these files accordingly.

```
- In the service file, set `--chain` to `mainnet` or `mumbai` accordingly
```

Save the changes in `/lib/systemd/system/heimdalld.service`.

- Make sure the chain is set correctly in `/var/lib/bor/config.toml` file. Open the file with following command `sudo vi /var/lib/bor/config.toml`
  - In the config file, set `chain` to `mainnet` or `mumbai` accordingly.
  - To enable Archive mode, you can optionally enable the following flags:

    ```
    gcmode "archive"

    [jsonrpc]
      [jsonrpc.ws]
        enabled = true
        port = 8546
        corsdomain = ["*"]
    ```

  Save the changes in `/var/lib/bor/config.toml`.

Starting the Sentry Node

You will first start the Heimdall service. Once the Heimdall service syncs, you will start the Bor service.

Reload service files

Reloading service files to make sure all changes to service files are loaded correctly.

```
sudo systemctl daemon-reload
```

Starting the Heimdall service

Start the Heimdall services:

```
sudo service heimdalld start
```

Check the Heimdall service logs:

```
journalctl -u heimdalld.service -f
```

:::note

In the logs, you may see the following errors:

- `Stopping peer for error`
- `MConnection flush failed`
- `use of closed network connection`

These logs mean that one of the nodes on the network refused a connection to your node. Wait for your node to crawl more nodes on the network; you do not need to do anything to address these errors.

:::

Check the Heimdalld logs:

```
journalctl -u heimdalld.service -f
```

Check the sync status of Heimdall:

```
curl localhost:26657/status
```

In the output, the `catching_up` value is:

- `true` — the Heimdall service is syncing.
- `false` — the Heimdall service is fully synced.

Wait for the Heimdall service to sync fully.

Starting the Bor service

Once the Heimdall service syncs, start the Bor service.

Start the Bor service:

```
sudo service bor start
```

Check the Bor service logs:

```
journalctl -u bor.service -f
```

Installing packages on the Validator Node

Follow the same [installation steps]() on validator node.

Configuring the Validator Node

:::note

To complete this section, you must have an RPC endpoint of your fully synced Ethereum mainnet node ready.

:::

:::caution

Bor and Heimdall 0.3.0 uses standardized paths for configuration files and chain data. If you have existing config files and chain data on your node, please jump directly to **Migration section** to learn about migrating configs and data to standardized file locations.

:::

Configure Heimdall

Log in to the remote validator machine.

Initialize heimdall configs

```
# For mainnet
sudo -u heimdall heimdalld init --chain=mainnet --home /var/lib/heimdall

# For testnet
sudo -u heimdall heimdalld init --chain=mumbai --home /var/lib/heimdall
```

Open the Heimdall configuration file for editing:

```
vi /var/lib/heimdall/config/config.toml
```

In `config.toml`, change the following:

- `moniker` — any name. Example: `moniker = "my-validator-node"`.
- `pex` — set the value to `false` to disable the peer exchange. Example: `pex = false`.
- `private_peer_ids` — comment out the value to disable it. Example: `# private_peer_ids = ""`.

  To get the node ID of Heimdall on the sentry machine:

  1. Log in to the sentry machine.
  2. Run `heimdalld tendermint show-node-id`.

Example: `persistent_peers = "sentry_machineNodeID@sentry_instance_ip:26656"`

- `prometheus` — set the value to `true` to enable the Prometheus metrics. Example: `prometheus = true`.

Save the changes in `config.toml`.

Open for editing `vi /var/lib/heimdall/config/heimdall-config.toml`.

In `heimdall-config.toml`, change the following:

- `eth_rpc_url` — an RPC endpoint for a fully synced Ethereum mainnet node or testnet node, i.e Infura. `eth_rpc_url =<insert Infura or any full node RPC URL to Ethereum>`

Example: `eth_rpc_url = "https://nd-123-456-789.p2pify.com/60f2a23810ba11c827d3da642802412a"`

Save the changes in `heimdall-config.toml`.

Configuring Bor

In `/var/lib/bor/config.toml`, add the following:

```
[p2p]
    [p2p.discovery]
        static-nodes = ["<replace with enode://validator_machine_enodeID@validator_machine_ip:30303>"]
```

To get the node ID of Bor on the sentry machine:

1. Log into the sentry machine.
2. Run `bor bootnode -node-key /var/lib/bor/data/bor/nodekey -dry-run`.

Example output:
`"enode://410e359736bcd3a58181cf55d54d4e0bbd6db2939c5f548426be7d18b8fd755a0ceb730fe5cf7510c6fa6f0870e388277c5f4c717af66d53c440feedffb`

Example content of static node field in `/var/lib/bor/config.toml`:

```
[p2p]
    [p2p.discovery]
        static-nodes =
["enode://410e359736bcd3a58181cf55d54d4e0bbd6db2939c5f548426be7d18b8fd755a0ceb730fe5cf7510c6fa6f0870e388277c5f4c717af66d53c440feedf
```

Save the changes in `/var/lib/bor/config.toml`.

Setting the Owner and Signer Key

On Polygon, it is recommended that you keep the owner and signer keys different.

- Signer — the address that signs the checkpoint transactions. The recommendation is to keep at least 1 ETH on the signer address.
- Owner — the address that does the staking transactions. The recommendation is to keep the MATIC tokens on the owner address.

Generating a Heimdall private key

You must generate a Heimdall private key only on the validator machine. Do not generate a Heimdall private key on the sentry machine.

To generate the private key, run:

```
heimdallcli generate-validatorkey ETHEREUM_PRIVATE_KEY
```

where

- ETHEREUM_PRIVATE_KEY — your Ethereum wallet address.

This will generate `priv_validator_key.json`. Move the generated JSON file to the Heimdall configuration directory:

```
mv ./priv_validator_key.json /var/lib/heimdall/config
```

Generating a Bor keystore file

You must generate a Bor keystore file only on the validator machine. Do not generate a Bor keystore file on the sentry machine.

To generate the private key, run:

```
heimdallcli generate-keystore ETHEREUM_PRIVATE_KEY
```

where

- ETHEREUM_PRIVATE_KEY — your Ethereum private key.

When prompted, set up a password to the keystore file.

This will generate a `UTC-<time>-<address>` keystore file.

Move the generated keystore file to the Bor configuration directory:

```
mv ./UTC-<time>-<address> /var/lib/bor/keystore/
```

Add password.txt

Make sure to create a `password.txt` file then add the Bor keystore file password right in the `/var/lib/bor/password.txt` file.

Add your Ethereum address

Open for editing `vi /etc/matic/metadata`.

In `metadata`, add your Ethereum address. Example: `VALIDATOR_ADDRESS=0xca67a8D767e45056DC92384b488E9Af654d78DE2`.

Save the changes in `metadata`.

Configure service files for bor and heimdall

After successfully installing Bor and Heimdall through packages, their service file could be found under `/lib/systemd/system`, and Bor's config file could be found under `/var/lib/bor/config.toml`. You will need to check and modify these files accordingly.

- Make sure the chain is set correctly in `/lib/systemd/system/heimdalld.service` file. Open the file with following command `sudo vi /lib/systemd/system/heimdalld.service`
  - In the service file, set `--chain` to `mainnet` or `mumbai` accordingly
  - Add `--bridge --all` to the heimdall command line for validator, example:

    ```
    ExecStart=/usr/local/bin/heimdalld start --home /var/lib/heimdall \
      --chain=mainnet \
      --bridge --all \
      --rest-server
    ```

  Save the changes in `/lib/systemd/system/heimdalld.service`.

- Make sure the chain is set correctly in `/var/lib/bor/config.toml` file. Open the file with following command `sudo vi /var/lib/bor/config.toml`
  - In the config file, set `chain` to `mainnet` or `mumbai` accordingly.
  - Enable validator flags, example:

    ```
    [miner]
      mine = true
      gaslimit = 20000000
      gasprice = "30000000000"
      etherbase = "VALIDATOR ADDRESS"

    [accounts]
      allow-insecure-unlock = true
      password = "/var/lib/bor/password.txt"
      unlock = ["VALIDATOR ADDRESS"]
    ```

  Save the changes in `/var/lib/bor/config.toml`.

Starting the Validator Node

At this point, you must have:

- The Heimdall service on the sentry machine syncs and is running.
- The Bor service on the sentry machine running.
- The Heimdall service and the Bor service on the validator machine configured.
- Your owner and signer keys configured.

Reload service files

Reloading service files to make sure all changes to service files are loaded correctly.

```
sudo systemctl daemon-reload
```

Starting the Heimdall service

You will now start the Heimdall service on the validator machine. Once the Heimdall service syncs, you will start the Bor service on the validator machine.

:::note

The Heimdall service takes several days to sync from scratch fully.

Alternatively, you can use a maintained snapshot, which will reduce the sync time to a few hours. For detailed instructions, see [Snapshot Instructions for Heimdall and Bor](#).

For snapshot download links, see [Polygon Chains Snapshots](#).

:::

Start the Heimdall services:

```
sudo service heimdalld start
```

Check the Heimdall service logs:

```
journalctl -u heimdalld.service -f
```

Check the sync status of Heimdall:

```
curl localhost:26657/status
```

In the output, the `catching_up` value is:

- `true` — the Heimdall service is syncing.
- `false` — the Heimdall service is synced.

Wait for the Heimdall service to fully sync.

Starting the Bor service

Once the Heimdall service on the validator machine syncs, start the Bor service on the validator machine.

Start the Bor service:

```
sudo service bor start
```

Check the Bor service logs:

```
journalctl -u bor.service -f
```

Seed nodes and bootnodes

- Heimdall seed nodes:

```
moniker=<enter unique identifier>

# Mainnet:
seeds="2a53a15ffc70ad41b6876ecbe05c50a66af01e20@3.211.248.31:26656,6f829065789e5b156cbbf076f9d133b4d7725847@3.212.183.151:26

# Testnet:
seeds="e72c0466a02ea43b2198bd3a9454b87a3ef0d77e@54.147.31.250:26656,b302d1ddb21102e794b524d05152a7834da05bd1@34.226.134.117:
```

- Bootnodes:

```
# Mainnet:
bootnode
["enode://0cb82b395094ee4a2915e9714894627de9ed8498fb881cec6db7c65e8b9a5bd7f2f25cc84e71e89d0947e51c76e85d0847de848c7782b13c02

# Testnet:
bootnodes
["enode://320553cda00dfc003f499a3ce9598029f364fbb3ed1222fdc20a94d97dcc4d8ba0cd0bfa996579dcc6d17a534741fb0a5da303a90579431259
```

Health Checks with the Community

Now that your sentry and validator nodes are in sync and running, head over to Discord and ask the community to health-check your nodes.

:::note

As validators, it's mandatory to always have a check of the signer address. If the ETH balance reaches below 0.5 ETH then it should be refilled. Avoiding this will push out nodes from submitting checkpoint transactions.

:::

Next Steps: Staking

Now that you have your sentry and validator nodes are health-checked, proceed to the Staking guide to start backing the network.
--- id: technical-faqs title: Technical FAQs description: Frequently asked questions related to running a Validator on the Polygon network. keywords: - docs - polygon - polygon wiki - faqs - technical slug: technical-faqs image: https://wiki.polygon.technology/img/polygon-logo.png ---

1. Are the private keys same for Heimdall and Bor keystore?

Yes, the private key used for generating Validator keys and Bor Keystore is the same. The private key used in this instance is your Wallet's ETH address where your Polygon testnet tokens are stored.

2. List of Common Commands

We currently have an easy to dive-in list for you for the Linux packages. We will keep updating this list regularly for more convenience.

For Linux packages

####A. Where to find heimdall genesis file

`$CONFIGPATH/heimdall/config/genesis.json`

####B. Where to find heimdall-config.toml

`/etc/heimdall/config/heimdall-config.toml`

####C. Where to find config.toml

`/etc/heimdall/config/config.toml`

####D. Where to find heimdall-seeds.txt

`$CONFIGPATH/heimdall/heimdall-seeds.txt`

####E. Start Heimdall

`$ sudo service heimdalld start`

####F. Start Heimdall rest-server

`$ sudo service heimdalld-rest-server start`

####G. Start Heimdall bridge-server

`$ sudo service heimdalld-bridge start`

####H. Heimdall logs

`/var/log/matic-logs/`

####I. Where to find Bor genesis file

`$CONFIGPATH/bor/genesis.json`

####J. Start Bor

`sudo service bor start`

####K Check heimdall logs

`tail -f heimdalld.log`

####L. Check Heimdall rest-server

`tail -f heimdalld-rest-server.log`

####M. Check Heimdall bridge logs

`tail -f heimdalld-bridge.log`

####N. Check bor logs

`tail -f bor.log`

####O. Kill Bor process

For linux:

1. `ps -aux | grep bor` . Get the PID for Bor and then run the following command.
2. `sudo kill -9 PID`

For Binaries:

Go to `CS-2003/bor` and then run, `bash stop.sh`

3. Error: Failed to unlock account (0x...) No key for given address or file

This error occurs because the path for the password.txt file is incorrect. You can follow the below steps to rectify this:

This error occurs because the path for the password.txt and Keystore file is incorrect. You can follow the below steps to rectify this:

1. Copy the bor keystore file to

   /etc/bor/dataDir/keystore

2. And password.txt to

   /etc/bor/dataDir/

3. Make sure you have added correct address in `/etc/bor/metadata`

For Binaries:

1. Copy the Bor keystore file to:

   `/var/lib/bor/keystore/`

2. And password.txt to

   `/var/lib/bor/password.txt`

4. Error: Wrong Block.Header.AppHash. Expected xxxx

This usually occurs due to an incorrect installation of heimdall. You can follow the steps below to rectify this:

run

```
```heimdalld unsafe-reset-all```
```

and start Heimdall services again. You can refer to this guide - https://docs.polygon.technology/docs/maintain/validate/run-validator

5. From where do I create the API key?

You can access this link: https://infura.io/register . Make sure that once you have setup your account and project, you copy the API key for Ropsten and not Mainnet.

Mainnet is selected by default.

6. Heimdall isn't working. I'm getting a Panic error

Actual Error: My heimdalld isn't working. In the log the first line is: panic: Unknown db_backend leveldb, expected either goleveldb or memdb or fsdb

Change the config to `goleveldb` in `config.toml` .

7. How do I delete remnants of Heimdall and Bor?
```

If you want to delete remnants of Heimdall and Bor then you can run the following commands Bor:

For Linux package:

`$ sudo dpkg -i matic-bor`

And delete Bor Directory:

`$ sudo rm -rf /etc/bor`

For Binaries:

`$ sudo rm -rf /etc/bor`

And

`$ sudo rm /etc/heimdall`

8. How many validators can be active concurrently?

There will be upto 100 active validators at a time. We will bring in more participants if the limit is reached mid-way through the event as well. Note that active validators is mostly those whose uptime is high. Participants with high downtime will be forced out.

9. How much should I stake?

"stake-amount" and "heimdall-fee-amount" - how much it should be?

A minimum of 10 Matic tokens is required for the stake amount whereas heimdall fee should be greater than 10. For example, your stake amount is 400 then the heimdall fee should be 20. We suggest to keep the Heimdall fee as 20.

However, please note that the values entered in stake amount and heimdal-fee-amount should be entered in 18 decimals

For example,

```
heimdallcli stake --staked-amount 40000000000000000000  --fee-amount 1000000000000000000 --validator
0xf8d1127780b89f167cb4578935e89b8ea1de774f
```

10. I was selected to become a validator but my ETH address was incorrect. What do I do?

If you have access to the ETH address that you submitted earlier then you can transfer the Test tokens from that account to the current account. And then you can initiate your process of setting up your nodes.

If you don't have access to that ETH address, we won't be transferring you tokens separately. You can re-register in the form again with the correct ETH address.

11. I'm getting an error starting the bridge

Error: Object "start" is unknown, try "bridge help". Is it still ok to ignore this?

Check "which bridge" - if it's `/usr/sbin/bridge` you're not running the right "bridge" program.

Try `~/go/bin/bridge` instead `(or $GOBIN/bridge)`

12. I'm getting dpkg error

Error: "dpkg: error processing archive matic-heimdall_1.0.0_amd64.deb (--install): trying to overwrite '/heimdalld-rest-server.service', which is also in package matic-node 1.0.0"

This occurs mainly because of a previous installation of Polygon on your machine. To resolve you can run:

`sudo dpkg -r matic-node`

13. I'm not clear on which Private Key should I add when I generate validator key

The Private key to be used is your Wallet's ETH address where your Polygon testnet Tokens are stored. You can complete the setup with one public-private key pair tied to the address submitted on the form.

14. Is there a way to know if Heimdall is synced?

You can run the following command to check it:

`$ curl [http://localhost:26657/status](http://localhost:26657/status)`

Check the value of catching_up. If it is false then the node is all synced up.

15. What if someone become a Top 10 staker, how he will receive his MATIC reward at the end?

Stage 1 rewards are not based on stake. Please refer to https://blog.matic.network/counter-stake-stage-1-stake-on-the-beach-full-details-matic-network/ for the reward details. Participants with high stake don't automatically qualify for a reward in this stage.

16. What should be my heimdall version?

To check your Heimdall version you can simply run:

`heimdalld version`

The correct version of Heimdall for stage 1 should be `heimdalld version is beta-1.1-rc1-213-g2bfd1ac`

17. What values should I add in the stake amount and fee amount?

A minimum of 10 Matic tokens is required for the stake amount whereas heimdall fee should be greater than 10. For example, your stake amount is 400 then the heimdall fee should be 20. We suggest to keep the Heimdall fee as 20.

However, please note that the values entered in stake amount and heimdal-fee-amount should be entered in 18 decimals

For example,

```
heimdallcli stake --staked-amount 40000000000000000000  --fee-amount 1000000000000000000 --validator
0xf8d1127780b89f167cb4578935e89b8ea1de774f
```

18. Whats the difference between `/var/lib/heimdall` and `/etc/heimdall?`

`/var/lib/heimdall` is the heimdall dir when you use the binary installation method. `/etc/heimdall` is for the Linux package installation method.

19. When I make the stake transaction, I'm getting "Gas Exceeded" error

This error may occur because of the stake or fee amount format. The values entered during the stake command need to have 18 decimals.

However, please note that the values entered in stake amount and heimdal-fee-amount should be entered in 18 decimals

For example,

```
heimdallcli stake --staked-amount 40000000000000000000  --fee-amount 1000000000000000000 --validator
0xf8d1127780b89f167cb4578935e89b8ea1de774f
```

20. When will I get a chance to become a Validator?

We are progressively adding validators throughout the course of Stage 1 event. We will be releasing a list of new external validators gradually. This list will be announced on the Discord channel.

21. Where can I find Heimdall account info location?

For binaries:

```
/var/lib/heimdall/config folder
```

For Linux package:

```
/etc/heimdall/config
```

22. Which file do I add the API key in?

Once you have created the API key you need to add the API key in `heimdall-config.toml` file.

23. Which file do I add the persistent_peers?

You can add the persistent_peers in the following file:

```
/var/lib/heimdall/config/config.toml
```

24. “Did you reset Tendermint without resetting your application's data?”

In such a case you can reset heimdall config data and try running the installation again.

```
$ heimdalld unsafe-reset-all
$ rm -rf $HEIMDALLDIR/bridge
```

25. Error: Unable to unmarshall config Error 1 error(s) decoding

Error: `* '' has invalid keys: clerk_polling_interval, matic_token, span_polling_interval, stake_manager_contract, stakinginfo_contract`

This occurs mostly because when there are typos, or some missing parts or an old config file which is still a remnant. You will need to clear all the remnants and then try setting it up again.

26. To stop Heimdall and Bor services

For Linux packages:

Stop Heimdall: `sudo service heimdalld stop`

Stop Bor: `sudo service bor stop` or

1. `ps -aux | grep bor`. Get the PID for Bor and then run the following command.
2. `sudo kill -9 PID`

For Binaries:

Stop Heimdall: `pkill heimdalld`

Stop Bridge: `pkill heimdalld-bridge`

Stop Bor: Go to CS-2001/bor and then run, `bash stop.sh`

27. To remove Heimdall and Bor directories

For Linux packages: Delete Heimdall: `sudo rm -rf /etc/heimdall/*`

Delete Bor: `sudo rm -rf /etc/bor/*`

For Binaries:

Delete Heimdall: `sudo rm -rf /var/lib/heimdall/`

Delete Bor: `sudo rm -rf /var/lib/bor`

28. What to do when you get "Wrong Block.Header.AppHash." error

This error usually occurs due to Infura requests getting exhausted. When you setup a node on Polygon, you add an Infura Key to the Config file (Heimdall). By default you are allowed 100k Requests per day, if this limit is crossed, then you would face such problems. To resolve this you can create a new API key and add it to the `config.toml` file.

:::tip Stay in the know

Keep up with the latest node and validator updates from the Polygon team and the community by subscribing to the Polygon notification groups.

:::

id: validator-commission-operations title: Commission Operations description: Set up and change your validator commissions keywords: - docs - matic - polygon - validator - commission operations slug: validator-commission-operations image: https://wiki.polygon.technology/img/polygon-logo.png

import useBaseUrl from '@docusaurus/useBaseUrl';

You can set up and change your commission as a validator.

A validator is entitled to charge any commission rate. The minimum commission would be 0% and the maximum commission would be 100% of the rewards earned.

You are allowed to change the commission rates as many times as you would like.

As a validator, it is one of your responsibilities to inform the community on commission changes. See Validator Responsibilities.

Set up your commission rate as a validator

You set up the commission rate as part of your initial validator staking process. See Validator Staking Operations.

Change your commission rate

You can change your commission rate.

1. With your owner address, login to the staking dashboard.
2. On your profile, click **Edit Profile**.
3. In the **Commission** field, provide your new commission rate.

Once you have confirmed and signed the transaction your commission rate will be set.

Once the commission is updated, there is a cool down period of 80 checkpoints.

id: validator-node-system-requirements title: System Requirements description: System requirements to run a validator node keywords: - docs - matic - polygon - prerequisites - requirements slug: validator-node-system-requirements image: https://wiki.polygon.technology/img/polygon-logo.png

The system requirements listed in this section are both for the Sentry node and the Validator node.

The **minimum** system requirements mean you can run the nodes but the setup is not future-proof.

The **recommended** system requirements mean the nodes are future-proof. There is, however, no upper limit to future-proofing your nodes.

You must always run the sentry node and the validator node on separate machines.

Minimum system requirements

- RAM: 32 GB
- CPU: 8-core
- Storage: 2.5 TB SSD

:::info

For Amazon Web Services (AWS), the equivalent of the minimum requirements instances are, **with unlimited credits selected**:

- For Sentry: **c5.2xlarge**
- For Validator node: **c5.4xlarge**

:::

Recommended system requirements

- RAM: 64 GB
- CPU: 16-core
- Storage: 5 TB SSD
- Bandwidth: 1 Gbit/s

:::info

For Amazon Web Services (AWS), the equivalent of the recommended requirements instance is **m5d.4xlarge**.

For OVH, the equivalent of the recommended requirements instance is **infra-3**.

For network, expect 3-5 TB of data transferred per month.

:::

id: validator-performance-overview title: Validator Performance Framework sidebar_label: Validator Performance Framework description: Learn about the validator performance metrics that help validators self-regulate. keywords: - docs - wiki - polygon - validator - performance - behavior - dashboard - staking slug: validator-performance image: https://wiki.polygon.technology/img/polygon-logo.png

import useBaseUrl from '@docusaurus/useBaseUrl';

Validator performance framework

This is a guide to help users understand the validator performance parameters the community agreed to in PIP-4.

These parameters and the related health statuses help provide information about validator performance against a defined performance benchmark.

Parameters

A validator's performance is measured based on checkpoints it signed over a monitoring period. Performance is measured on a rolling basis at each new checkpoint to provide an objective figure. This figure is then measured against a benchmark of the total network performance in the monitoring period, as detailed below.

- Monitoring Period ("MP") = previous 700 checkpoints, updated every new checkpoint.
- Take % of checkpoints signed by each validator in the MP and find the median.
- Multiply the median average by an agreed multiple = Performance Benchmark ("PB").
- At each checkpoint, calculate the % of checkpoints signed in the MP by single validators and measure against the PB.

Performance Benchmark

To facilitate the transition, there will be a slightly lower benchmark around the first two months while validators become accustomed to the parameters.

- PB1 → 95% of the median average of the last 700 checkpoints signed by the validator set (first 2,800 checkpoints))
- PB2 → 98% of the median average of last checkpoints signed by validator set (continues thereafter)

Deficient Validator Process

- If validator <PB in the MP → Grace Period 1 ("GP1").
- If validator is in GP and <PB after 700 checkpoints → Notice of Deficiency ("NOD"), validator enters into Grace Period 2 ("GP2").
- If validator is in GP2 and <PB after 700 checkpoints → Final Notice ("FN"), the validator will be unstaked per PIP-4.

Each GP is an additional 700 checkpoints, allowing a validator to bring their performance back above the PB. If the deficiency is corrected within the GP, there will be no further action. Failure to match, or out perform the performance benchmark, at the end of GP2 will result in a public NOD that will be displayed on the staking dashboard.

The validator will have a 700 checkpoint period to correct the deficiency in GP2. If the deficiency is fixed within the NOD period, then no further action will occur. However, the NOD would remain public. Failure to improve after GP2 would result in the issuance of an FN of the community's intent to implement a forced exit procedure by offboarding the validator from the network by unbonding their stake.

:::tip Health Status

You can quickly check a validator's health status, whether it is **Healthy**, on **Grace Period** or **Final Notice** on the **All Validators** table or on each validator page.

:::

Public Notices

The Public Notices page shows the recent notices and messages sent to the community of validators.

This is how a Notice message should look like:

Forced Unstaking

The unstaking of the deficient validator would be done as follows:

Call the `ForceUnstake` function in Polygon Commitchain Contract: 0xFa7D2a996aC6350f4b56C043112Da0366a59b74c

id: validator-staking-operations title: How to Stake on Polygon PoS Mainnet sidebar_label: How to Stake on PoS Mainnet description: Learn how to stake as a validator on the Polygon Network keywords: - docs - matic - polygon - stake - claim - unstake slug: validator-staking-operations image: https://wiki.polygon.technology/img/polygon-logo.png

import useBaseUrl from '@docusaurus/useBaseUrl';

Prerequisites

Full node set up

Your validator node fully set up and synced. See also: Run a Validator Node

Account setup

On your validator node, check that the account is set up. To check, run the following command **on the validator node**:

```
heimdalld show-account
```

Your output should appear in the following format:

```
{
    "address": "0x6c468CF8c9879006E22EC4029696E005C2319C9D",
    "pub_key":
"0x04b12d8b2f6e3d45a7ace12c4b2158f79b95e4c28ebe5ad54c439be9431d7fc9dc1164210bf6a5c3b8523528b931e772c86a307e8cff4b725e6b4a77d21417b
}
```

This will display your address and public key for your validator node. Note that **this address must match with your signer address on Ethereum**.

Show private key

This step is optional.

On your validator node, check that the private key is correct. To check, run the following command **on the validator node**:

```
heimdalld show-privatekey
```

The following output should appear:

```
{
    "priv_key": "0x*********************************************************"
}
```

Stake on Polygon

You can stake on Polygon using the validator dashboard.

Stake using the staking dashboard

1. Access the validator dashboard.
2. Log in with your wallet. MetaMask is the recommended wallet. You have to make sure that you login using the same address where your MATIC tokens are present.
3. Click **Become a Validator**. You will be asked to set up your node. If you haven't already set up your node by now, you will need to do so, else if you proceed ahead you will receive an error when you attempt to stake.
4. On the next screen, add your validator details, the commission rate, and the staking amount.
5. Click **Stake Now**.

Once the transaction is completed you will have staked successfully to become a validator. You will be asked thrice to confirm the transaction.

- Approve Transaction — this will approve your stake transaction.
- Stake — This will confirm your stake transaction.

- Save —ß This will save your validator details.

:::note

For the changes to take effect on the staking dashboard, it requires a minimum of 12 block confirmations.

:::

Check the balance

To check the balance of your address:

```
heimdallcli query auth account SIGNER_ADDRESS --chain-id CHAIN_ID
```

where

- SIGNER_ADDRESS — your signer address.
- CHAIN_ID — the Polygon mainnet chain ID with the client prefix: `heimdall-137`.

The following output should appear:

```
address: 0x6c468cf8c9879006e22ec4029696e005c2319c9d
coins:
- denom: matic
amount:
    i: "100000000000000000000"
accountnumber: 0
sequence: 0
```

Claim rewards as a validator

Once you are set up and staked as a validator, you will earn rewards for performing validator duties. When you perform validator duties dutifully, you get rewarded.

To claim rewards you can go to your validator dashboard.

You will see two buttons on your profile:

- Withdraw Reward
- Restake Reward

Withdraw Reward

As a validator, you earn rewards as long as you are performing your validator duties correctly.

Clicking **Withdraw Reward** will get your rewards back to your wallet.

The dashboard will update after 12 block confirmations.

Restake Reward

Restaking your rewards is an easy way to increase your stake as a validator.

Clicking **Restake Reward** will restake your reward and increase your stake.

The dashboard will update after 12 block confirmations.

id: polygon-architecture title: Architecture Overview sidebar_label: Overview description: Introduction to the architecture of Polygon PoS blockchain. keywords: - architecture - polygon - pos - wiki - research image: https://wiki.polygon.technology/img/polygon-logo.png

import useBaseUrl from '@docusaurus/useBaseUrl';

The Polygon PoS Network has a three-layer architecture:

- **Ethereum layer** — a set of contracts on the Ethereum mainnet.
- **Heimdall layer** — a set of proof-of-stake Heimdall nodes running parallel to the Ethereum mainnet, monitoring the set of staking contracts deployed on the Ethereum mainnet and committing the Polygon Network checkpoints to the Ethereum mainnet. Heimdall is based on Tendermint.
- **Bor layer** — a set of block-producing Bor nodes shuffled by Heimdall nodes. Bor is based on Go Ethereum.

<img src={useBaseUrl("img/staking/architecture.png")} />

Currently, developers can use PoS for state transitions for which predicates have been written, such as ERC20, ERC721, asset swaps, or other custom predicates. they can use PoS.

To enable the PoS mechanism on our platform, a set of **staking** management contracts are deployed on Ethereum, and a set of incentivized validators running **Heimdall** and **Bor** nodes. Ethereum is the first basechain Polygon supports, but Polygon intends to offer support for additional basechains to enable an interoperable decentralized Layer 2 blockchain platform based on community suggestions and consensus.

<img src={useBaseUrl("img/matic/Architecture.png")} />

Staking Contracts

To enable the Proof of Stake (PoS) mechanism on Polygon, the system employs a set of staking management contracts on the Ethereum mainnet.

The staking contracts implement the following features:

- Anyone can stake MATIC tokens on the staking contracts on the Ethereum mainnet and join the system as a validator.
- Earn staking rewards for validating state transitions on the Polygon Network.
- Save checkpoints on the Ethereum mainnet.

The PoS mechanism also acts as a mitigation to the data unavailability problem for the Polygon sidechains.

Heimdall

Heimdall is the proof of stake validation layer that handles the aggregation of blocks produced by Bor into a Merkle tree and publishes the Merkle root periodically to the root chain. The periodic publishing of snapshots of Bor is called checkpoints.

1. Validates all the blocks since the last checkpoint.
2. Creates a Merkle tree of the block hashes.
3. Publishes the Merkle root hash to the Ethereum mainnet.

Checkpoints are important for two reasons:

1. Providing finality on the root chain.
2. Providing proof of burn in withdrawal of assets.

An overview of the process:

- A subset of active validators from the pool is selected to act as block producers for a span. These block producers are responsible for creating blocks and broadcasting the created blocks on the network.
- A checkpoint includes the Merkle root hash of all blocks created during any given interval. All nodes validate the Merkle root hash and attach their signature to it.
- A selected proposer from the validator set is responsible for collecting all signatures for a particular checkpoint and committing the checkpoint on the Ethereum mainnet.
- The responsibility of creating blocks and proposing checkpoints is variably dependent on a validator's stake ratio in the overall pool.

More details on Heimdall are available on the Heimdall architecture guide.

Bor

Bor is Polygon's block producer layer - the entity responsible for aggregating transactions into blocks. Currently, it is a basic Geth implementation with custom changes done to the consensus algorithm.

Block producers are a subnet of the validators and are periodically shuffled via committee selection on Heimdall in durations termed as a `span` in Polygon. Blocks are produced at the **Bor** node, and the VM is EVM-compatible. Blocks produced on Bor are also validated periodically by Heimdall nodes, and a checkpoint consisting of the Merkle tree hash of a set of blocks on Bor is committed to Ethereum periodically.

More details are available on the Bor architecture guide.

Resources

- Bor Architecture
- Heimdall Architecture
- Checkpoint Mechanism

---

id: getting-started title: Get Started with Polygon PoS sidebar_label: Quick Start description: "Build your next blockchain app on Polygon." keywords: - docs - matic - polygon - build on polygon - blockchain - introduction - how to launch dapp - dapps - ethereum image: https://wiki.polygon.technology/img/polygon-logo.png

This guide will introduce you to the Polygon PoS ecosystem. You'll find links to valuable resources and websites that will bring you up to speed on building, not only on Polygon but also on general blockchain application development.

:::tip Stay in the know

Keep up with the latest builder updates from the Polygon team and the community by subscribing to the Polygon notification groups.

:::

Building on Polygon

If you are an Ethereum developer, you are already a Polygon developer. Simply switch to the Polygon RPC and get started. All the tools you are familiar with on the Ethereum blockchain are supported on Polygon by default, such as Truffle, Remix, and Web3js.

You can deploy decentralized applications to either Polygon Mumbai Testnet or the Mainnet. The Polygon Mumbai Testnet connects with the Ethereum Goërli Testnet, which acts as its ParentChain. You can find all the network-related details in the network documentation.

Wallets

To interact with the Polygon Network, you need to have an Ethereum-based wallet because Polygon runs on Ethereum Virtual Machine (EVM). You can choose to set up a Metamask or Arkane Wallet. More on wallet-related information and why you need one can be found in our wallet documentation.

Smart Contracts

Polygon supports many services you can use to test, compile, debug, and deploy decentralized applications onto the Polygon Network. These include deployment using thirdweb, Alchemy, Chainstack, QuickNode, Remix, Truffle, Hardhat, and Replit.

Connecting to Polygon

You can add Polygon to MetaMask or directly use Arkane, which allows you to connect to Polygon using RPC.

:::note Directory of publicly available PoS RPC endpoints

Check out a list of public endpoints for the Polygon PoS network at Alchemy's Chain Connect resource and Chainlist.

:::

In order to connect with the Polygon network to read blockchain information, we recommend using the Alchemy SDK.

```javascript
// Javascript
// Setup: npm install alchemy-sdk
const { Alchemy, Network } = require("alchemy-sdk");

const settings = {
  apiKey: "demo", // Can replace with your API Key from https://www.alchemy.com
  network: Network.MATIC_MAINNET, // Can replace with MATIC_MUMBAI
};

const alchemy = new Alchemy(settings);

async function main() {
  const latestBlock = await alchemy.core.getBlockNumber();
  console.log("The latest block number is", latestBlock);
}

main();
```

Building a new dApp on Polygon?

Decentralized applications (dApps) act as the bridge between users and their data privacy on the blockchain. The increasing number of dApps validates their usefulness within the blockchain ecosystem, solving challenges like executing transactions between two participants without the need for central authority via smart contracts.

Suppose you have no prior experience building decentralized applications (dApps). In that case, the below-mentioned resources will give you a head start on the tools required to build, debug, and deploy dApps on the Polygon Network.

- Full Stack dApp: Tutorial Series
- Web3.js
- Ethers.js
- thirdweb
- Remix
- Truffle
- Metamask
- Arkane
- Develop a dApp using Fauna, Polygon and React

Already have a dApp?

If you already have a decentralized application (dApp) and are looking for a platform to help you scale efficiently, then you are at the right place because Polygon allows you to:

1. **Easily migrate from Ethereum Virtual Machine (EVM) based chain**: Polygon prides itself in being the ultimate Layer-2 scaling solution for Ethereum. You don't have to worry about the underlying architecture while moving or deploying your dApps to the Polygon Network as long as it is EVM-compatible.
2. **Use Polygon as a faster transaction layer**: Deploying your dApp to the Polygon Mainnet allows you to leverage Polygon as a faster transaction layer for your dApp. Additionally, you can get your tokens mapped by us. You can join our technical discussions group on Telegram to learn more.

Side Note

If this is overwhelming, that’s alright! You can jump right into the action and start hacking. Here are some notes before you start diving into resources, repositories, and docs:

1. **Beware the cost of being on the bleeding edge**: Like typical niche programming, dApps and blockchain development moves very quickly. While researching, you may find complex code repositories, 404s on a documentation site, or even no documentation. Use that opportunity to reach out to us via any social media channel.
2. **The learning curve may be daunting, but the barrier to entry is low**: The community is very open and welcoming! Projects welcome pull requests from outsiders and resolve any blockers actively. We’re working on creating a better world and contribution in any form is appreciated. We’ll be grateful to onboard you into this amazing Web3 ecosystem.

:::info Stay Updated

Decentralized application development encourages network decentralization. Follow our social media handles for more insights and updates about the Polygon ecosystem. You can find the links to all the Polygon communities here.

:::

General Resources

:::caution Content disclaimer

Please view the third-party content disclaimer here.

:::

Basics of Web3 Development

- What is Ethereum?
- Mastering Ethereum
- Full-Stack dApp Tutorial Series
- Web3 Developer Stack
- Deploy a Smart Contract using REMIX
- How to Create a Token (ERC20)?
- Integrating IPFS with Ethereum
- Hello World Smart Contract
- How to Create an NFT
- Lens Protocol | How to Create a Decentralized Twitter
- Truffle Tutorial
- Dapp University

Learn more about blockchain

- Polygon's Architecture and Security
- Integrating Webhooks
- Archive Nodes with Debug namespace

External documentation

- thirdweb's docs
- Alchemy's Polygon Docs
- QuickNode's Polygon Docs
- ConsenSys Academy Developer Course
- Infura Docs
- Truffle Docs
- Geth Docs
- Remix Docs
- OpenZeppelin Docs
- Cloudflare Docs

Useful tutorials

- Alchemy University
- CryptoZombies
- Ethernaut
- Capture the Ether
- Road to Web 3

Additional resources

- Video Tutorials Library
- Polygon dApps
- Writings by the Team
- Matic Tools
- FAQs

---

id: commit-chain-multisigs title: PoS Mainnet Multisigs sidebar_label: Multisigs description: Information about active multisigs keywords: - docs - polygon - matic - multisig - address image: https://wiki.polygon.technology/img/polygon-logo.png

import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem';

:::note Capabilities

The primary function of the multisigs is to enable contract upgrades in the [current] early phases of development.

As the contracts secured by the multisigs become battle-tested, Polygon will introduce the following optimizations and changes:

- move from multisigs to governance-controlled proxies;
- introduce timelocks;
- eventually remove all multisigs.

To be clear, the current multisigs cannot censor transactions like bridge transactions.

:::

Multisig Address	**5/9 multisig** **0xFa7D2a996aC6350f4b56C043112Da0366a59b74c**
Purpose	To upgrade POS and staking contracts on Ethereum*

Chain	Ethereum
Rights	• Update staking contracts for staking optimisations, features and upgradations • Be able to fix unexpected and unforeseen bugs on POS contracts
Signatories	• Quickswap • Curve • Polygon • Horizon Games • Cometh

Multisig Address	**5/8 multisig 0x355b8E02e7F5301E6fac9b7cAc1D6D9c86C0343f**
Purpose	To to be able to update "custom" childerc20s if required on Polygon Commitchain*
Chain	Polygon Commitchain
Rights	• Ability to upgrade "custom child contracts" (and not the fx portal created standard tokens mentioned below)
Signatories	• Quickswap • Curve • Polygon • Horizon Games • Cometh

Multisig Address	****Permissionless Mapping of standard ChildERC20 tokens (No Multisig Required) ****
Purpose	Use FxPortal supports permissionless token mapping of standard ChildERC20 for any ERC20 token on Ethereum
Chain	Permissionless
Rights	Permissionless
Signatories	Permissionless

Multisig Address	**4/8 Multisig 0x424bDE99FCfB68c5a1218fd3215caFfD031f19C4**
Purpose	• To enable the mapping of "custom" child ERC20s with Mainnet contract. • 60% of Dapps deployed on Polygon "want" custom child ERC20 token, which needs to be mapped with supervision. • Eventually this can go to Governance but this will vastly increase the time for Dapp teams who want custom ERC20. • You can create standard child ERC20s yourselves using Fx portal, its permissionless, no mapping required
Chain	Ethereum
Rights	Can only do mapping, doesn't have any access to Child token, no deposit/withdrawal rights whatsoever
Signatories	• Polygon

***To be moved to governance. Exploring Aave's governance contracts and Compound's time lock contracts**

id: delegation title: Delegation via Validator Shares sidebar_label: Delegation description: Delegation via Validator Shares keywords: - polygon wiki - docs - polygon - delegation - validator shares image: https://wiki.polygon.technology/img/polygon-logo.png

Polygon supports delegation via validator shares. By using this design, it is easier to distribute rewards and slash with scale (thousands of delegators) on Ethereum contracts without much computation.

Delegators delegate by purchasing shares of a finite pool from validators. Each validator will have their own validator share token. Let's call these fungible tokens `VATIC` for a validator `A`. As soon as a user delegates to a validator `A`, they will be issued `VATIC` based on an exchange rate of `MATIC/VATIC` pair. As users accrue value the exchange rate indicates that they can now withdraw more `MATIC` for each `VATIC` and when users get slashed, users withdraw less `MATIC` for their `VATIC`.

Note that `MATIC` is a staking token. A delegator needs to have `MATIC` tokens to participate in the delegation.

Initially, a delegator `D` buys tokens from validator `A` specific pool when `1 MATIC per 1 VATIC`.

When a validator gets rewarded with more `MATIC` tokens, new tokens are added to the pool. Let's say with the current pool of `100 MATIC` tokens, `10 MATIC` rewards are added to the pool. But since the total supply of `VATIC` tokens didn't change due to rewards, the exchange rate becomes `1 MATIC per 0.9 VATIC`. Now, delegator `D` gets more `MATIC` for the same shares.

`VATIC` : Validator specific minted validator share tokens (ERC20 tokens)

Technical Specification

```
uint256 public validatorId; // Delegation contract for validator
uint256 public validatorRewards; // accumulated rewards for validator
uint256 public commissionRate; // validator's cut %
uint256 public validatorDelegatorRatio = 10; // to be implemented/used

uint256 public totalStake;
uint256 public rewards; // rewards for pool of delegation stake
uint256 public activeAmount; // # of tokens delegated which are part of active stake
```

Exchange rate is calculated as below:

```
ExchangeRate = (totalDelegatedPower + delegatorRewardPool) / totalDelegatorShares
```

Methods and Variables

buyVoucher

```
function buyVoucher(uint256 _amount) public;
```

- Transfer the `_amount` to stakeManager and update the timeline data structure for active stake.
- `updateValidatorState` is used to update timeline DS.
- `Mint` delegation shares using current `exchangeRate` for `_amount`.
- `amountStaked` is used to keep track of active stake of each delegator in order to calculate liquid rewards.

sellVoucher

```
function sellVoucher() public;
```

- Using current `exchangeRate` and number of shares to calculate total amount (active stake + rewards).
- `unBond` active stake from validator and transfer rewards to delegator, if any.
- Must remove active stake from timeline using `updateValidatorState` in stakeManger.
- `delegators` mapping is used to keep track of stake in withdrawal period.

withdrawRewards

```
function withdrawRewards() public;
```

- For a delegator, calculate the rewards and transfer, and depending upon `exchangeRate` burn count of shares.
- Example: if a delegator owns 100 shares and exchange rate is 200 so rewards are 100 tokens, transfer 100 tokens to delegator. Remaining stake is 100 so using exchange rate 200, now it is worth 50 shares. So burn 50 shares. Delegator now has 50 shares worth 100 tokens (which he initially staked / delegated).

reStake

Restake can work in two ways: delegator can buy more shares using `buyVoucher` or reStake rewards.

```
function reStake() public;
```

Above function is used to reStake rewards. The number of shares aren't affected because `exchangeRate` is the same; so just the rewards are moved into active stake for both validator share contract and stakeManager timeline.

`getLiquidRewards` is used for calculating accumulated rewards i.e., delegator owns 100 share and exchange rate is 200, so rewards are 100 tokens. Move 100 tokens into active stake, since exchange rate is still same number of share will also remain same. Only difference is that now 200 tokens are considered into active stake and can't be withdrawn immediately (not a part of liquid rewards).

Purpose of reStaking is that since delegator's validator has now more active stake and they will earn more rewards for that so will the delegator.

unStakeClaimTokens

```
function unStakeClaimTokens()
```

Once withdrawal period is over, delegators who've sold their shares can claim their MATIC tokens. Must transfer tokens to user.

updateCommissionRate

```
function updateCommissionRate(uint256 newCommissionRate)
        external
        onlyValidator
```

- Updates commission % for the validator.

updateRewards

```
function updateRewards(uint256 reward, uint256 checkpointStakePower, uint256 validatorStake)
        external
        onlyOwner
        returns (uint256)
```

When a validator gets rewards for submitting checkpoint, this function is called for disbursements of rewards between validator and delegators.

id: genesis-contracts title: Genesis Contracts description: Genesis contracts list on Polygon keywords: - docs - matic - polygon - genesis contracts image: https://wiki.polygon.technology/img/polygon-wiki.png

import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem';

Here you will find a list of contracts deployed on Polygon together with their initial address, that is, their location on the blockchain.

<Tabs defaultValue="mainnet" values={[{ label: 'Polygon-Mainnet', value: 'mainnet', }, { label: 'Mumbai-Testnet', value: 'mumbai', },] }>

Parent Chain: Goerli

Contracts	Address
BytesLib	0xde5807d201788dB32C38a6CE0F11d31b1aeB822a
Common	0x84Dc17F28658Bc74125C7E82299992429ED34c12
ECVerify	0xccd1d8d16F462f9d281024CBD3eF52BADB10131C
Merkle	0xCD87Be2Df3de01EA23666c97104613ec252300E8
MerklePatriciaProof	0x3a0Db8fa2805DEcd49cCAa839DaC15455498EDE2
PriorityQueue	0xD26361204b8e4a4bb16668bfE7A1b9106AD17140
RLPEncode	0xDE0D18799a20f29d9618f8DDbf4c2b029FAdc491
RLPReader	0xA5e463c187E53da5b193E2beBca702e9fEeA3738
SafeMath	0x1bEb355BE0577E61870C4c57DAaa6e2129dd0604
Governance	0x03Ac67D03A06571A059F20425FFD1BEa300d98C2
GovernanceProxy	0xAcdEADEE4c054A86F5b1e8705126b30Ec999899B
Registry	0xeE11713Fe713b2BfF2942452517483654078154D
RootChain	0x77f32d6c7bf3639b2f59c6843420e80e9e3a86af
RootChainProxy	0x2890bA17EfE978480615e330ecB65333b880928e
ValidatorShareFactory	0x5737AD9095AB4d55FeE7F972ea7F86734695E3c1
StakingInfo	0x29C40836C17f22d16a7fE953Fb25DA670C96d69E
StakingNFT	0x532c7020E0F3666f9440B8B9d899A9763BCc5dB7
StakeManager	0xc1b9b1ff63ef14502f7c6c8e9f5eed47654695ae
StakeManagerProxy	0x00200eA4Ee292E253E6Ca07dBA5EdC07c8Aa37A3
SlashingManager	0xDD17DE137c7Cc288E022fE95a3B398C94BDd5b83
ValidatorShare	0xa7957328580692f6440948c3c479a9cde17de206
StateSender	0xEAa852323826C71cd7920C3b4c007184234c3945
DepositManager	0x20339c5Ea91D680E681B9374Fc0a558D5b96a026
DepositManagerProxy	0x7850ec290A2e2F40B82Ed962eaf30591bb5f5C96
WithdrawManager	0xb075cdda944d4e1ff19f2201cdc3a440a11d4710
ExitNFT	0xE2Ab047326B38e4DDb6791551e8d593D30E02724
WithdrawManagerProxy	0x2923C8dD6Cdf6b2507ef91de74F1d5E0F11Eac53
ERC20Predicate	0x39c1e715316a1acbce0e6438cf62edf83c111975
ERC721Predicate	0x473cb675c9214f79dee10948443509c441a678e7

[Child Chain: Mumbai]

Contracts	Address
ChildChain	0x1EDd419627Ef40736ec4f8ceffdE671a30803c5e

Parent Chain: Ethereum Mainnet

Contracts	Address
BytesLib	0x1d21fACFC8CaD068eF0cbc87FdaCdFb20D7e2417
Common	0x31851aAf1FA4cC6632f45570c2086aDcF8B7BD75
ECVerify	0x71d91a8988D81617be53427126ee62471321b7DF
Merkle	0x8b90C7633F1f751E19E76433990B1663c625B258
MerklePatriciaProof	0x8E51a119E892D3fb324C0410F11f39F61dec9DC8
PriorityQueue	0x61AdDcD534Bdc1721c91740Cf711dBEcE936053e
RLPEncode	0x021c2Bf4d2941cE3D593e07317EC355937bae495
RLPReader	0xD75f1d6A8A7Dc558A65c2f30eBF876DdbeE035a2
SafeMath	0x96D358795782a73d90F2ed2d505aB235D197ca05
Governance	0x98165b71cdDea047C0A49413350C40571195fd07
GovernanceProxy	0x6e7a5820baD6cebA8Ef5ea69c0C92EbbDAc9CE48
Registry	0x33a02E6cC863D393d6Bf231B697b82F6e499cA71
RootChain	0x536c55cFe4892E581806e10b38dFE8083551bd03
RootChainProxy	0x86E4Dc95c7FBdBf52e33D563BbDB00823894C287
ValidatorShareFactory	0xc4FA447A0e77Eff9717b09C057B40570813bb642
StakingInfo	0xa59C847Bd5aC0172Ff4FE912C5d29E5A71A7512B
StakingNFT	0x47Cbe25BbDB40a774cC37E1dA92d10C2C7Ec897F
StakeManager	0xd6f5c46d4e1a02f9d145cee41d2f8af30d8d2d76
StakeManagerProxy	0x5e3Ef299fDDf15eAa0432E6e66473ace8c13D908
SlashingManager	0x01F645DcD6C796F6BC6C982159B32fAaaebdC96A
ValidatorShare	0x01d5dc56ad4206bb0c132d834644d57f51fed5ec
StateSender	0x28e4F3a7f651294B9564800b2D01f35189A5bFbE
DepositManager	0xd505C3822C787D51d5C2B1ae9aDB943B2304eB23
DepositManagerProxy	0x401F6c983eA34274ec46f84D70b31C151321188b
WithdrawManager	0x017C89Ca4Bda3D66cC65E3d20DD95432258201Ca
ExitNFT	0xDF74156420Bd57ab387B195ed81EcA36F9fABAca
WithdrawManagerProxy	0x2A88696e0fFA76bAA1338F2C74497cC013495922
ERC20Predicate	0x158d5fa3ef8e4dda8a5367decf76b94e7effce95
ERC721Predicate	0x54150f44c785d412ec262fe895cc3b689c72f49b

Child Chain: Polygon Mainnet

Contracts	Address
ChildChain	0xD9c7C4ED4B66858301D0cb28Cc88bf655Fe34861
---	
id: stakingmanager	
title: Staking Manager	
description: Staking Manager is the main contract for handling validator-related activities on Polygon network.	
keywords:	

- docs
- Staking Manager
- polygon
- wiki
- validator image: https://wiki.polygon.technology/img/polygon-logo.png

---

import useBaseUrl from '@docusaurus/useBaseUrl';

For the Polygon's Proof of Security based consensus, all the ⅔+1 proof verification and handling of staking, rewards are executed on the Ethereum smart contract. The whole design follows this philosophy of doing less on the Mainnet contract. It does information verification and pushes all the computation-heavy operations to L2 (read about Heimdall).

Stakers are divided into **validators**, **delegators**, and **watchers** (for fraud reporting).

StakeManager is the main contract for handling validator related activities like `checkPoint` signature verification, reward distribution, and stake management. Since the contract is using **NFT ID** as a source of ownership, change of ownership and signer won't affect anything in the system.

:::tip

From one Ethereum address, a **Staker can only be a validator or delegator** (it's just a design choice, no hard reasons).

:::

Validator Admissions / Replacement

Admissions

At present, there are no open validator slots available on Polygon PoS. There is also a waitlist to become a validator. In the future, if slots become available, validators may apply to be considered and removed off of the waitlist.

Replacement

PIP4 introduced the concept of showcasing validator performance for community visibility. If a validator is in an unhealthy state for an extended period of time as outlined in PIP4, they are off-boarded from the network. The validator slot is then made available to those coming off of the waitlist.

:::info

Currently, Phase 2 of PART C in PIP4 is being implemented. This is where the community decides on validator prospect evaluation criteria. In time, this exercise will produce an application and admissions process.

:::

Methods and Variables

:::caution Slashing Implementation

`jail`, `unJail`, and `slash` functions are not used currently as part of the slashing implementation.

:::

validatorThreshold

It stores the maximum number of validators accepted by the system, also called slots.

AccountStateRoot

- For various accounting done on Heimdall for validators and delegator, account root is submitted while submitting the `checkpoint`.
- accRoot is used while `claimRewards` and `unStakeClaim`.

stake / stakeFor

```
function stake(
    uint256 amount,
    uint256 heimdallFee,
    bool acceptDelegation,
    bytes calldata signerPubkey
) public;

function stakeFor(
    address user,
    uint256 amount,
    uint256 heimdallFee,
    bool acceptDelegation,
    bytes memory signerPubkey
) public;
```

- Allows anyone with amount (in MATIC tokens) greater than `minDeposit`, if `currentValidatorSetSize` is less then `validatorThreshold`.
- Must transfer `amount+heimdallFee`, puts validator into auction period for an auctionInterval (more in Auction section).
- `updateTimeLine` updates special timeline data structure, which keeps track of active validators and active stake for given epoch / checkpoint count.
- One unique `NFT` is minted on each new `stake` or `stakeFor` call, which can be transferred to anyone but can be owned 1:1 Ethereum address.
- `acceptDelegation` set to true if validators want to accept delegation, `ValidatorShare` contract is deployed for the validator.

unstake

- Remove validator from validator set in next epoch (only valid for current checkpoint once called `unstake`)
- Remove validator's stake from timeline data structure, update count for validator's exit epoch.
- If validator had delegation on, collect all rewards and lock delegation contract for new delegations.

unstakeClaim

```
function unstakeClaim(uint256 validatorId) public;
```

- After `unstaking`, validators are put into withdrawal period so that they can be slashed, if any fraud found after `unstaking`, for past frauds.
- Once `WITHDRAWAL_DELAY` period is served, validators can call this function and do settlement with `stakeManager` (get rewards if any, get staked tokens back, burn NFT, etc).

restake

```
function restake(uint256 validatorId, uint256 amount, bool stakeRewards) public;
```

- Allows validators to increase their stake by putting new amount or rewards or both.
- Must update timeline (amount) for active stake.

withdrawRewards

```
function withdrawRewards(uint256 validatorId) public;
```

This method allows validators to withdraw accumulated rewards, must consider getting rewards from delegation contract if validator accepts delegation.

updateSigner

```
function updateSigner(uint256 validatorId, bytes memory signerPubkey) public
```

This method allows validators to update signer address (which is used to validate blocks on Polygon blockchain and checkpoint signatures on `stakeManager`).

topUpForFee

```
function topUpForFee(uint256 validatorId, uint256 heimdallFee) public;
```

Validators can top-up their balance for Heimdall fee by invoking this method.

claimFee

```
function claimFee(
        uint256 validatorId,
        uint256 accumSlashedAmount,
        uint256 accumFeeAmount,
        uint256 index,
        bytes memory proof
    ) public;
```

This method is used to withdraw fees from Heimdall. `accountStateRoot` is updated on each checkpoint, so that validators can provide proof of inclusion in this root for account on Heimdall and withdraw fee.

Note that `accountStateRoot` is re-written to prevent exits on multiple checkpoints (for old root and save accounting on `stakeManager`). `accumSlashedAmount` is unused at the moment and will be used for slashing on Heimdall if needed.

StakingNFT

Standard ERC721 contract with few restrictions like one token per user and minted in sequential manner.

startAuction

```
function startAuction(
    uint256 validatorId, /**  auction for validator */
    uint256 amount /**  amount greater then old validator's stake */
    ) external;
```

In order to start a bid or bid higher on already running auction, this function is used. Auction period runs in cycles like `(auctionPeriod--dynasty)--(auctionPeriod--dynasty)--(auctionPeriod--dynasty)` so it **must check for correct auction period**.

`perceivedStakeFactor` is used to calculate exact factor*old stake (note currently it is by default 1 WIP for picking the function). **Must check for auction from last auction period if any still going on** (one can choose to not call `confirmAuction` in order to get their capital out in the next auction). Normally continuous english auction is going on in a `auctionPeriod`.

confirmAuctionBid

```
function confirmAuctionBid(
        uint256 validatorId,
        uint256 heimdallFee, /** for new validator */
        bool acceptDelegation,
```

```
        bytes calldata signerPubkey
    ) external
```

- **Must check that this is not an auctionPeriod.**
- If last bidder is owner of `validatorId`, behaviour should be similar to restake.
- In second case unStake `validatorId` and add new user as validator from next checkpoint, for the new user behaviour should be similar to stake/stakeFor.

checkSignatures

```
function checkSignatures(
        uint256 blockInterval,
        bytes32 voteHash,
        bytes32 stateRoot,
        bytes memory sigs
    ) public;
```

- Writes are meant only for RootChain contract when submitting checkpoints
- `voteHash` on which all validators sign (BFT ⅔+1 agreement)
- This function validates only unique sigs and checks for ⅔+1 power has signed on checkpoint root (inclusion in `voteHash` verification in RootChain contract for all data) `currentValidatorSetTotalStake` provides current active stake.
- Rewards are distributed proportionally to validator's stake. More on rewards in Rewards Distribution.

isValidator

Checks if a given validator is active validator for the current epoch.

Timeline Data Structure

```
struct State {
    int256 amount;
    int256 stakerCount;
}

mapping(uint256 => State) public validatorState;
```

<img src={useBaseUrl("img/staking_manager/staking_manager.png")} />

StakingInfo

Centralized logging contract for both validator and delegation events, includes few read only functions. You can check out the source code of the StakingInfo.sol contract on GitHub.

ValidatorShareFactory

A factory contract to deploy `ValidatorShare` contract for each validator who opt-in for delegation. You can check out the source code of the ValidatorShareFactory.sol contract on GitHub. --- id: mapped-tokens title: Mapped Tokens sidebar_label: Mapped tokens description: Examples about different mapped tokens on Polygon Mainnet keywords: - docs - matic - polygon - mainnet - mapped tokens image: https://wiki.polygon.technology/img/polygon-wiki.png

import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem';

This document provides examples of different mapped tokens on the Polygon PoS mainnet and Mumbai PoS testnet. The table in the document lists token name, bridge, parent chain address, and child chain for various tokens, including PoS-DAI, PoS-USDT, PoS-WETH, and PoS-ETH, among others. The documnet also includes instructions for getting a token mapped via mapper.polygon.technology.

:::tip Mapping

If you want to get your token mapped then please send your mapping request via mapper.polygon.technology.

:::

Below are tables that provide several examples of various mapped tokens for each network. It is important to note that these are not exhaustive lists.

<Tabs defaultValue="mainnet" values={[{ label: 'PoS mainnet', value: 'mainnet', }, { label: 'Mumbai testnet', value: 'mumbai', },] }>

Token Name	Bridge	Parent Chain Address: Goerli	Child Chain: MUMBAI
PoS-WETH	PoS	0x60D4dB9b534EF9260a88b0BED6c486fe13E604Fc	0xA6FA4fB5f76172d178d61B04b0ecd319C5d1C0aa
DummyERC20Token	PoS	0x655F2166b0709cd575202630952D71E2bB0d61Af	0xfe4F5145f6e09952a5ba9e956ED0C25e3Fa4c7F1
DummyERC721Token	PoS	0x084297B12F204Adb74c689be08302FA3f12dB8A7	0x757b1BD7C12B81b52650463e7753d7f5D0565C0e

DummyERC1155Token	PoS	0x2e3Ef7931F2d0e4a7da3dea950FF3F19269d9063	0xA07e45A987F19E25176c877d98388878622623FA

Token Name	Bridge	Parent Chain Address: Ethereum Mainnet	Child Chain: Polygon Network
PoS-WETH	PoS	0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE	0x7ceB23fD6bC0adD59E62ac25578270cFf1b9f619
PoS-USDC	PoS	0xa0b86991c6218b36c1d19d4a2e9eb0ce3606eb48	0x2791Bca1f2de4661ED88A30C99A7a9449Aa84174
PoS-COMP	PoS	0xc00e94cb662c3520282e6f5717214004a7f26888	0x8505b9d2254A7Ae468c0E9dd10Ccea3A837aef5c
PoS-LEND	PoS	0x80fB784B7eD66730e8b1DBd9820aFD29931aab03	0x313d009888329C9d1cf4f75CA3f32566335bd604
PoS-YFI	PoS	0x0bc529c00C6401aEF6D220BE8C6Ea1667F6Ad93e	0xDA537104D6A5edd53c6fBba9A898708E465260b6
PoS-USDT	PoS	0xdac17f958d2ee523a2206206994597c13d831ec7	0xc2132D05D31c914a87C6611C10748AEb04B58e8F
PoS-DAI	PoS	0x6b175474e89094c44da98b954eedeac495271d0f	0x8f3Cf7ad23Cd3CaDbD9735AFf958023239c6A063
PoS-BUSD	PoS	0x4fabb145d64652a948d72533023f6e7a623c7c53	0xdAb529f40E671A1D4bF91361c21bf9f0C9712ab7
PoS-MANA	PoS	0x0f5d2fb29fb7d3cfee444a200298f468908cc942	0xA1c57f48F0Deb89f569dFbE6E2B7f46D33606fD4
PoS-WBTC	PoS	0x2260fac5e5542a773aa44fbcfedf7c193bc2c599	0x1BFD67037B42Cf73acF2047067bd4F2C47D9BfD6
PoS-0xBTC	PoS	0xb6ed7644c69416d67b522e20bc294a9a9b405b31	0x71B821aa52a49F32EEd535fCA6Eb5aa130085978
PoS-KIWI	PoS	0x2BF91c18Cd4AE9C2f2858ef9FE518180F7B5096D	0x578360AdF0BbB2F10ec9cEC7EF89Ef495511ED5f
PoS-DUST	PoS	0xbca3c97837a39099ec3082df97e28ce91be14472	0x556f501CF8a43216Df5bc9cC57Eb04D4FFAA9e6D
PoS-SPN	PoS	0x20f7a3ddf244dc9299975b4da1c39f8d5d75f05a	0xeAb9Cfb094db203e6035c2e7268A86DEbeD5BD14
DummyERC20Token	PoS	0xf2F3bD7Ca5746C5fac518f67D1BE87805a2Be82A	0xeFfdCB49C2D0EF813764B709Ca3c6fe71f230E3e
DummyERC721Token	PoS	0x71B821aa52a49F32EEd535fCA6Eb5aa130085978	0x6EBEAC13f6403D19C95b6B75008B12fd21a93Aab
DummyERC1155Token	PoS	0x556f501CF8a43216Df5bc9cC57Eb04D4FFAA9e6D	0xA0c68C638235ee32657e8f720a23ceC1bFc77C77
WDEV	PoS	0x4a5df63b0c37b38515e4ee51baf40edd420bf7d5	0xa5577d1cec2583058a6bd6d5deac44797c205701

--- id: default-ports title: Default Ports For Nodes sidebar_label: Ports description: List of default ports used across Polygon PoS nodes. keywords: - docs - polygon - matic - port - port management - default ports image: https://wiki.polygon.technology/img/polygon-wiki.png ---

Here is a list of default ports used across Polygon nodes:

Bor

Name	Port	Tags	description
Network listening port	30303	public	Network listening port. Bor uses this port to connect to peers and sync
RPC server	8545	can-be-public, internal	RPC port to send transaction and get data from Bor. Heimdall uses this port to get Bor headers for checkpoints
WS server	8546	can-be-public, internal	Websocket port
Graphql server	8547	can-be-public, internal	Graphql port
Prometheus server	9091	can-be-public, monitoring	Prometheus server APIs as datasource in Grafana. It can be mapped to 80/443 through nginx reverse proxy
Grafana server	3001	can-be-public, monitoring	Grafana web sever. It can be mapped to 80/443 through nginx reverse proxy
Pprof server	7071	internal, monitoring	Pprof server to collect metrics from Bor
UDP discovery	30301	can-be-public, internal	Bootnode default port (for peer discovery)

Heimdall

Name	Port	Tags	description
Network listening port	30303	public	Network listening port. Bor uses this port to connect to peers and sync
RPC server	8545	can-be-public, internal	RPC port to send transaction and get data from Bor. Heimdall uses this port to get Bor headers for checkpoints
WS server	8546	can-be-public, internal	Websocket port
Graphql server	8547	can-be-public, internal	Graphql port

Prometheus server	9091	can-be-public, monitoring	Prometheus server APIs as datasource in Grafana. It can be mapped to 80/443 through nginx reverse proxy
Grafana server	3001	can-be-public, monitoring	Grafana web sever. It can be mapped to 80/443 through nginx reverse proxy
Pprof server	7071	internal, monitoring	Pprof server to collect metrics from Bor
UDP discovery	30301	can-be-public, internal	Bootnode default port (for peer discovery)
id: rpc-endpoints			
title: Network Endpoints			
sidebar_label: Endpoints			
description: Network endpoints for Polyon PoS Mainnet and Mumbai Testnet.			
keywords:			

- docs
- polygon
- remote procedure call
- network endpoints
- rpcs
- http
- websocket
- wss image: https://wiki.polygon.technology/img/polygon-wiki.png

---

import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem';

This index guide contains network details for the Polygon Mumbai testnet and Polygon PoS Mainnet and lists their associated RPC and node endpoints.

<Tabs defaultValue="mainnet" values={[{ label: 'PoS Mainnet', value: 'mainnet', }, { label: 'Mumbai Testnet', value: 'mumbai', },] }>

Mumbai Testnet replicates the Polygon Mainnet and is used for testing. Users can obtain testnet tokens from the faucet. Testnet tokens are valueless and are different from value-bearing assets like MATIC. This allows developers or network maintainers to test configurations and experiment implementations.

Properties	Network Details
NetworkName	**Mumbai**
ParentChain	**Goërli**
chainId	`80001`
Gas Token	MATIC
Gas Station	Mumbai Gas Station (learn more here)
RPC Endpoint	https://rpc-mumbai.matic.today
Node Endpoint	wss://rpc-mumbai.matic.today
Heimdall API	https://heimdall-api-testnet.polygon.technology
Block Explorer	https://mumbai.polygonscan.com/

:::note More details

See the following **JSON data** containing network details.

:::

The native token of the Polygon PoS is MATIC and is used for gas.

Properties	Network Details
NetworkName	**Polygon**
ParentChain	**Ethereum**
chainId	`137`
Gas Token	MATIC
Gas Station	PolygonScan Gas Tracker (**recommended**) or Matic Network Gas Station (learn more here)
RPC Endpoint	https://polygon-rpc.com/
Node Endpoint	wss://rpc-mainnet.matic.network
Heimdall API	https://heimdall-api.polygon.technology

Block Explorer	https://polygonscan.com/

:::note More details

See the following **JSON data** containing network details.

:::

RPC API Methods

Developers can interact with on-chain data and send different types of transactions to the network by utilizing network endpoints. The APIs follow a JSON-RPC standard; JSON-RPC is a stateless, lightweight, remote procedure call (RPC) protocol that is commonly used when interacting with a blockchain network.

:::info Get started with RPC calls

Start by visiting the complete set of API documentation for standard **Polygon JSON-RPC calls**.

If you want to get started with API requests that require no setup, fix failing requests, or, explore new methods on the Polygon network, try out the **Composer App**.

:::

A user can also run their own node when interacting with the Polygon PoS chain or utilize one of the public endpoints provided by infrastructure and API service providers to connect to the network. Dagger is the best way to get realtime updates from the chain as it provides a way for your dApps and backend system to get blockchain events in realtime over a socket or websocket.

Infrastructure Providers

Public RPCs may have traffic or rate-limits depending on usage. You can sign up for a dedicated free RPC URL at the following:

- Alchemy
- Allnodes
- Ankr
- Blast (Bware Labs)
- BlockPI
- BlockSpaces
- Chainnodes
- Chainstack
- DataHub (Figment)
- GetBlock
- Infura
- MaticVigil
- Moralis
- NodeReal
- OnFinality
- Pocket Network
- QuickNode
- SettleMint
- Tatum
- WatchData
- NOWNodes
- Kriptonio
- Chain49

View all public endpoints on Alchemy's Chain Connect and Chainlist.

id: snapshot-instructions-heimdall-bor title: Heimdall and Bor Snapshots sidebar_label: Snapshots description: Heimdall and Bor snapshot Instructions for faster syncing. keywords: - docs - matic - polygon - binary - node - validator - sentry - heimdall - bor - snapshots image: https://wiki.polygon.technology/img/polygon-wiki.png

import useBaseUrl from '@docusaurus/useBaseUrl';

When setting up a new sentry, validator, or full node server, it is recommended that you use snapshots for faster syncing without having to sync over the network. Using snapshots will save you several days for both Heimdall and Bor.

:::tip

For the latest snapshot, please visit Polygon Chains Snapshots.

:::

Heimdall/Bor Snapshots

To begin, ensure that your node environment meets the **prerequisites** outlined here. Before starting any services, execute the shell script provided below. This script will download and extract the snapshot data, which allows for faster bootstrapping. In our example, we will be using an Ubuntu Linux m5d.4xlarge machine with an 8TB block device attached. To transfer the correct chaindata to your disk, follow these steps:

- When prompted, specify the network (mainnet or mumbai) and the client type (heimdall or bor).

  > *The script will automatically handle the download and extraction phases, optimizing disk space by deleting already extracted files.*

- Consider using a Screen session to prevent accidental interruptions during the chaindata download and extraction process.

```
#!/bin/bash

function validate_network() {
  if [[ "$1" != "mainnet" && "$1" != "mumbai" ]]; then
    echo "Invalid network input. Please enter 'mainnet' or 'mumbai'."
    exit 1
  fi
}

function validate_client() {
  if [[ "$1" != "heimdall" && "$1" != "bor" ]]; then
    echo "Invalid client input. Please enter 'heimdall' or 'bor'."
    exit 1
  fi
}

# ask user for network and client type
read -p "PoSV1 Network (mainnet/mumbai): " network_input
validate_network "$network_input"
read -p "Client Type (heimdall/bor): " client_input
validate_client "$client_input"
read -p "Directory to Download/Extract: " extract_dir_input

# set default values if user input is blank
network=${network_input:-mumbai}
client=${client_input:-heimdall}
extract_dir=${extract_dir_input:-"${client}_extract"}

# install dependencies and cursor to extract directory
sudo apt-get update -y
sudo apt-get install -y zstd pv aria2
mkdir -p "$extract_dir"
cd "$extract_dir"

# download compiled incremental snapshot files list
aria2c -x6 -s6 "https://snapshot-download.polygon.technology/$client-$network-incremental-compiled-files.txt"

# download all incremental files, includes automatic checksum verification per increment
aria2c -x6 -s6 -c --auto-file-renaming=false --max-tries=100 -i $client-$network-incremental-compiled-files.txt

# Don't extract if download failed
if [ $? -ne 0 ]; then
    echo "Download failed. Restart the script to resume downloading."
    exit 1
fi

# helper method to extract all files and delete already-extracted download data to minimize disk use
function extract_files() {
    compiled_files=$1
    while read -r line; do
        if [[ "$line" == checksum* ]]; then
            continue
        fi
        filename=`echo $line | awk -F/ '{print $NF}'`
        if echo "$filename" | grep -q "bulk"; then
            pv $filename | tar -I zstd -xf - -C . && rm $filename
        else
            pv $filename | tar -I zstd -xf - -C . --strip-components=3 && rm $filename
        fi
    done < $compiled_files
}

# execute final data extraction step
extract_files $client-$network-incremental-compiled-files.txt
```

Note: If experiencing intermittent aria2c download errors, try reducing concurrency as exampled here:

```
aria2c -c -m 0 -x6 -s6 -i heimdall-$network-incremental-compiled-files.txt --max-concurrent-downloads=1
```

Once the extraction is complete, ensure that you update the datadir configuration of your Heimdall or Bor client to point to the path where the extracted data is located. This ensures that the systemd services can correctly register the snapshot data when the client starts. If you wish to preserve the default client configuration settings, you can use symbolic links (symlinks).

For example, let's say you have mounted your block device at `~/snapshots` and have downloaded and extracted the chaindata for Heimdall into the directory `heimdall_extract`, and for Bor into the directory `bor_extract`. To ensure proper registration of the extracted data when starting the Heimdall or Bor systemd services, you can use the following sample commands:

```
# remove any existing datadirs for heimdall and bor
rm -rf /var/lib/heimdall/data
rm -rf /var/lib/bor/chaindata

# rename and setup symlinks to match default client datadir configs
mv ~/snapshots/heimdall_extract ~/snapshots/data
mv ~/snapshots/bor_extract ~/snapshots/chaindata
sudo ln -s ~/snapshots/data /var/lib/heimdall
sudo ln -s ~/snapshots/chaindata /var/lib/bor

# bring up clients with all snapshot data properly registered
sudo service heimdalld start
# wait for heimdall to fully sync then start bor
sudo service bor start
```

Recommended disk size guidance

Polygon Mumbai Testnet

Metric	Calculation Breakdown	Value
approx. compressed total	250 GB (bor) + 35 GB (heimdall)	285 GB
approx. data growth daily	10 GB (bor) + .5 GB (heimdall)	10.5 GB
approx. total extracted size	350 GB (bor) + 50 GB (heimdall)	400 GB
suggested disk size (2.5x buffer)	400 GB * 2.5 (natural chain growth)	1 TB

Polygon Mainnet

Metric	Calculation Breakdown	Value
approx. compressed total	1500 GB (bor) + 225 GB (heimdall)	1725 GB
approx. data growth daily	100 GB (bor) + 5 GB (heimdall)	105 GB
approx. total extracted size	2.1 TB (bor) + 300 GB (heimdall)	2.4 TB
suggested disk size (2.5x buffer)	2.4 TB * 2.5 (natural chain growth)	6 TB

---

id: json-rpc-bridge title: Bridge description: "List of Bridge JSON RPC commands for Polygon Edge." keywords: - docs - polygon - edge - json - rpc - commands

---

bridge_generateExitProof

Returns the proof for a given exit event. Used by users that want to exit the L2 chain and withdraw tokens to L1.

Parameters

exitID - ID of the exit event submitted by L2StateSender contract when the user wants to exit the L2 contract.

Returns

- **Object** - A proof object containing:
  - **Array of hashes** - representing the proof of membership of a given exit event on some checkpoint.
  - **Map** - containing a leaf index of given exit event in the Merkle tree, Exit event and block number in which checkpoint that contains a given exit event was submitted to the rootchain.

---

bridge_getStateSyncProof

Returns the proof for a given state sync event. Used by users (Relayer) when they want to execute a state change on the childchain (for example, depositing funds from L1 to L2).

Parameters

stateSyncID - ID of the state sync event submitted by StateSender contract.

Returns

- **Object** - A proof object containing:
  - **Array of hashes** - representing the proof of membership of a given state sync event on some commitment.
  - **Map** - containing the state sync event data.

---

id: json-rpc-debug title: Debug description: "List of Debug JSON RPC commands for Polygon Edge." keywords: - docs - polygon - edge - json - rpc - commands

import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem'; import {useState} from 'react';

debug_traceBlockByNumber

Executes all transactions in the block specified by number with a tracer and returns the tracing result.

Parameters

- **QUANTITY|TAG** - integer of a block number, or the string "latest"
- **Object** - The tracer options:
  - **enableMemory: Boolean** - (optional, default: false) The flag indicating enabling memory capture.
  - **disableStack: Boolean** - (optional, default: false) The flag indicating disabling stack capture.
  - **disableStorage: Boolean** - (optional, default: false) The flag indicating disabling storage capture.
  - **enableReturnData: Boolean** - (optional, default: false) The flag indicating enabling return data capture.
  - **timeOut: String** - (optional, default: "5s") The timeout for cancellation of execution.

Returns

Array - Array of trace objects with the following fields:

- **failed: Boolean** - the tx is successful or not
- **gas: QUANTITY** - the total consumed gas in the tx
- **returnValue: DATA** - the return value of the executed contract call
- **structLogs: Array** - the trace result of each step with the following fields:
  - **pc: QUANTITY** - the current index in bytecode
  - **op: String** - the name of current executing operation
  - **gas: QUANTITY** - the available gas ßin the execution
  - **gasCost: QUANTITY** - the gas cost of the operation
  - **depth: QUANTITY** - the number of levels of calling functions
  - **error: String** - the error of the execution
  - **stack: Array** - array of values in the current stack
  - **memory: Array** - array of values in the current memory
  - **storage: Object** - mapping of the current storage
  - **refund: QUANTITY** - the total of current refund value

Example

```
curl  https://rpc-endpoint.io:8545 -X POST -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"debug_traceBlockByNumber","params":["latest"],"id":1}'
```

debug_traceBlockByHash

Executes all transactions in the block specified by block hash with a tracer and returns the tracing result.

Parameters

- **DATA , 32 Bytes** - Hash of a block.
- **Object** - The tracer options. See debug_traceBlockByNumber for more details.

Returns

Array - Array of trace objects. See debug_traceBlockByNumber for more details.

Example

```
curl  https://rpc-endpoint.io:8545 -X POST -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"debug_traceBlockByHash","params":
["0xdc0818cf78f21a8e70579cb46a43643f78291264dda342ae31049421c82d21ae"],"id":1}'
```

debug_traceBlock

Executes all transactions in the block given from the first argument with a tracer and returns the tracing result.

Parameters

- **DATA** - RLP Encoded block bytes
- **Object** - The tracer options. See debug_traceBlockByNumber for more details.

Returns

Array - Array of trace objects. See debug_traceBlockByNumber for more details.

Example

```
curl  https://rpc-endpoint.io:8545 -X POST -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"debug_traceBlock","params":["0xf9...."],"id":1}'
```

debug_traceTransaction

Executes the transaction specified by transaction hash with a tracer and returns the tracing result.

Parameters

- **DATA , 32 Bytes** - Hash of a transaction.
- **Object** - The tracer options. See debug_traceBlockByNumber for more details.

Returns

Object - Trace object. See debug_traceBlockByNumber for more details.

Example

```
curl  https://rpc-endpoint.io:8545 -X POST -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"debug_traceTransaction","params":
["0xdc0818cf78f21a8e70579cb46a43643f78291264dda342ae31049421c82d21ae"],"id":1}'
```

debug_traceCall

Executes a new message call with a tracer and returns the tracing result.

Parameters

- **Object** - The transaction call object
  - **from: DATA, 20 Bytes** - (optional) The address the transaction is sent from.
  - **to: DATA, 20 Bytes** - The address the transaction is directed to.
  - **gas: QUANTITY** - (optional) Integer of the gas provided for the transaction execution. eth_call consumes zero gas, but this parameter may be needed by some executions.
  - **gasPrice: QUANTITY** - (optional) Integer of the gasPrice used for each paid gas
  - **value: QUANTITY** - (optional) Integer of the value sent with this transaction
  - **data: DATA** - (optional) Hash of the method signature and encoded parameters. For details see Ethereum Contract ABI in the Solidity documentation
- **QUANTITY|TAG** - integer block number, or the string "latest"
- **Object** - The tracer options. See debug_traceBlockByNumber for more details.

Returns

Object - Trace object. See debug_traceBlockByNumber for more details.

Example

```
curl  https://rpc-endpoint.io:8545 -X POST -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"debug_traceCall","params":[{"to": "0x1234", "data": "0x1234"}, "latest", {}],"id":1}'
```

---

id: json-rpc-eth title: Ethereum description: "List of ETH JSON RPC commands for Polygon Edge." keywords: - docs - polygon - edge - json - rpc - commands

import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem'; import {useState} from 'react';

eth_chainId

Returns the currently configured chain id, a value used in replay-protected transaction signing as introduced by EIP-155.

Parameters

- None

Returns

- **QUANTITY** - big integer of the current chain id.

Example

```
curl  https://rpc-endpoint.io:8545 -X POST -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"eth_chainId","params":[],"id":1}'
```

eth_syncing

Returns information about the sync status of the node

Parameters

- None

Returns

* **Boolean (FALSE)** - if the node isn't syncing (which means it has fully synced)

* **Object** - an object with sync status data if the node is syncing

- **startingBlock: QUANTITY** - The block at which the import started (will only be reset, after the sync reached his head)
- **currentBlock: QUANTITY** - The current block, same as eth_blockNumber
- **highestBlock: QUANTITY** - The estimated highest block

Example

```
curl  https://rpc-endpoint.io:8545 -X POST -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"eth_syncing","params":[],"id":1}'
```

eth_getBlockByNumber

Returns block information by number.

Parameters

- **QUANTITY|TAG** - integer of a block number, or the string "latest"
- **Boolean** - If true it returns the full transaction objects, if false only the hashes of the transactions.

Returns

Object - A block object, or null when no block was found:

- **number: QUANTITY** - the block number.
- **hash: DATA, 32 Bytes** - hash of the block.
- **parentHash: DATA, 32 Bytes** - hash of the parent block.
- **nonce: DATA, 8 Bytes** - hash of the generated proof-of-work.
- **sha3Uncles: DATA, 32 Bytes** - SHA3 of the uncles data in the block.
- **logsBloom: DATA, 256 Bytes** - the bloom filter for the logs of the block.
- **transactionsRoot: DATA, 32 Bytes** - the root of the transaction trie of the block.
- **stateRoot: DATA, 32 Bytes** - the root of the final state trie of the block.
- **receiptsRoot: DATA, 32 Bytes** - the root of the receipts trie of the block.
- **miner: DATA, 20 Bytes** - the address of the beneficiary to whom the mining rewards were given.
- **difficulty: QUANTITY** - integer of the difficulty for this block.
- **totalDifficulty: QUANTITY** - integer of the total difficulty of the chain until this block.
- **extraData: DATA** - the “extra data” field of this block.
- **size: QUANTITY** - integer the size of this block in bytes.
- **gasLimit: QUANTITY** - the maximum gas allowed in this block.
- **gasUsed: QUANTITY** - the total used gas by all transactions in this block.
- **timestamp: QUANTITY** - the unix timestamp for when the block was collated.
- **transactions: Array** - Array of transaction objects, or 32 Bytes transaction hashes depending on the last given parameter.
- **uncles: Array** - Array of uncle hashes.

Example

```
curl  https://rpc-endpoint.io:8545 -X POST -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"eth_getBlockByNumber","params":["latest", true],"id":1}'
```

eth_getBlockByHash

Returns block information by hash.

Parameters

- **DATA , 32 Bytes** - Hash of a block.
- **Boolean** - If true it returns the full transaction objects, if false only the hashes of the transactions.

Returns

Object - A block object, or null when no block was found:

- **number: QUANTITY** - the block number.
- **hash: DATA, 32 Bytes** - hash of the block.
- **parentHash: DATA, 32 Bytes** - hash of the parent block.
- **nonce: DATA, 8 Bytes** - hash of the generated proof-of-work.
- **sha3Uncles: DATA, 32 Bytes** - SHA3 of the uncles data in the block.
- **logsBloom: DATA, 256 Bytes** - the bloom filter for the logs of the block.
- **transactionsRoot: DATA, 32 Bytes** - the root of the transaction trie of the block.
- **stateRoot: DATA, 32 Bytes** - the root of the final state trie of the block.

- **receiptsRoot: DATA, 32 Bytes** - the root of the receipts trie of the block.
- **miner: DATA, 20 Bytes** - the address of the beneficiary to whom the mining rewards were given.
- **difficulty: QUANTITY** - integer of the difficulty for this block.
- **totalDifficulty: QUANTITY** - integer of the total difficulty of the chain until this block.
- **extraData: DATA** - the “extra data” field of this block.
- **size: QUANTITY** - integer the size of this block in bytes.
- **gasLimit: QUANTITY** - the maximum gas allowed in this block.
- **gasUsed: QUANTITY** - the total used gas by all transactions in this block.
- **timestamp: QUANTITY** - the unix timestamp for when the block was collated.
- **transactions: Array** - Array of transaction objects, or 32 Bytes transaction hashes depending on the last given parameter.
- **uncles: Array** - Array of uncle hashes.

Example

```
curl  https://rpc-endpoint.io:8545 -X POST -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"eth_getBlockByHash","params":
["0xdc0818cf78f21a8e70579cb46a43643f78291264dda342ae31049421c82d21ae",false],"id":1}'
```

eth_blockNumber

Returns the number of the most recent block.

Parameters

None

Returns

- **QUANTITY** - integer of the current block number the client is on.

Example

```
curl  https://rpc-endpoint.io:8545 -X POST -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"eth_blockNumber","params":[],"id":1}'
```

eth_gasPrice

Returns the current price of gas in wei. If minimum gas price is enforced by setting the `--price-limit` flag, this endpoint will return the value defined by this flag as minimum gas price.

Parameters

None

Returns

- **QUANTITY** - integer of the current gas price in wei.

Example

```
curl  https://rpc-endpoint.io:8545 -X POST -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"eth_gasPrice","params":[],"id":1}'
```

eth_getBalance

Returns the balance of the account of the given address.

Parameters

- **DATA, 20 Bytes** - address to check for balance.
- **QUANTITY|TAG** - integer block number, or the string "latest"

Returns

- **QUANTITY** - integer of the current balance in wei.

Example

```
curl  https://rpc-endpoint.io:8545 -X POST -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"eth_getBalance","params":["0x407d73d8a49eeb85d32cf465507dd71d507100c1", "latest"],"id":1}'
```

eth_sendRawTransaction

Creates new message call transaction or a contract creation for signed transactions.

Parameters

- **DATA** - The signed transaction data.

Returns

- **DATA, 32 Bytes** - the transaction hash, or the zero hash if the transaction is not yet available.

Example

```
curl  https://rpc-endpoint.io:8545 -X POST -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"eth_sendRawTransaction","params":
["0xd46e8dd67c5d32be8d46e8dd67c5d32be8058bb8eb970870f072445675058bb8eb970870f072445675"],"id":1}'
```

eth_getTransactionByHash

Returns the information about a transaction requested by transaction hash.

Parameters

- **DATA, 32 Bytes** - hash of a transaction

Returns

Object - A transaction object, or null when no transaction was found:

- **blockHash: DATA, 32 Bytes** - hash of the block where this transaction was in.
- **blockNumber: QUANTITY** - block number where this transaction was in.
- **from: DATA, 20 Bytes** - address of the sender.
- **gas: QUANTITY** - gas provided by the sender.
- **gasPrice: QUANTITY** - gas price provided by the sender in Wei.
- **hash: DATA, 32 Bytes** - hash of the transaction.
- **input: DATA** - the data send along with the transaction.
- **nonce: QUANTITY** - the number of transactions made by the sender prior to this one.
- **to: DATA, 20 Bytes** - address of the receiver. null when its a contract creation transaction.
- **transactionIndex: QUANTITY** - integer of the transactions index position in the block.
- **value: QUANTITY** - value transferred in Wei.
- **v: QUANTITY** - ECDSA recovery id
- **r: DATA, 32 Bytes** - ECDSA signature r
- **s: DATA, 32 Bytes** - ECDSA signature s

Example

```
curl  https://rpc-endpoint.io:8545 -X POST -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"eth_getTransactionByHash","params":
["0x88df016429689c079f3b2f6ad39fa052532c56795b733da78a91ebe6a713944b"],"id":1}'
```

eth_getTransactionReceipt

Returns the receipt of a transaction by transaction hash.

Note That the receipt is not available for pending transactions.

Parameters

- **DATA, 32 Bytes** - hash of a transaction

Returns

Object - A transaction receipt object, or null when no receipt was found:

- **transactionHash : DATA, 32 Bytes** - hash of the transaction.
- **transactionIndex: QUANTITY** - integer of the transactions index position in the block.
- **blockHash: DATA, 32 Bytes** - hash of the block where this transaction was in.
- **blockNumber: QUANTITY** - block number where this transaction was in.
- **from: DATA, 20 Bytes** - address of the sender.
- **to: DATA, 20 Bytes** - address of the receiver. null when its a contract creation transaction.
- **cumulativeGasUsed : QUANTITY** - The total amount of gas used when this transaction was executed in the block.
- **gasUsed : QUANTITY** - The amount of gas used by this specific transaction alone.
- **contractAddress : DATA, 20 Bytes** - The contract address created, if the transaction was a contract creation, otherwise null.
- **logs: Array** - Array of log objects, which this transaction generated.
- **logsBloom: DATA, 256 Bytes** - Bloom filter for light clients to quickly retrieve related logs.

It also returns either :

- **root : DATA 32 bytes** - post-transaction stateroot (pre Byzantium)
- **status: QUANTITY** - either 1 (success) or 0 (failure)

Example

```
curl  https://rpc-endpoint.io:8545 -X POST -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"eth_getTransactionReceipt","params":
```

```
["0xb903239f8543d04b5dc1ba6579132b143087c68db1b2168786408fcbce568238"],"id":1}'
```

eth_getTransactionCount

Returns the number of transactions sent from an address.

Parameters

- **DATA, 20 Bytes** - address.
- **QUANTITY|TAG** - integer block number, or the string "latest"

Returns

- **QUANTITY** - integer of the number of transactions send from this address.

Example

```
curl  https://rpc-endpoint.io:8545 -X POST -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"eth_getTransactionCount","params":["0x407d73d8a49eeb85d32cf465507dd71d507100c1","latest"],"id":1}'
```

eth_getBlockTransactionCountByNumber

Returns the number of transactions in a block matching the given block number.

Parameters

- **QUANTITY|TAG** - integer of a block number, or the string "latest"

Returns

- **QUANTITY** - integer of the number of transactions in this block.

Example

```
curl  https://rpc-endpoint.io:8545 -X POST -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"eth_getBlockTransactionCountByNumber","params":["latest"],"id":1}'
```

eth_getLogs

Returns an array of all logs matching a given filter object.

Parameters

Object - The filter options:

- **fromBlock: QUANTITY|TAG** - (optional, default: "latest") Integer block number, or "latest" for the last mined block
- **toBlock: QUANTITY|TAG** - (optional, default: "latest") Integer block number, or "latest" for the last mined block
- **address: DATA|Array, 20 Bytes** - (optional) Contract address or a list of addresses from which logs should originate.
- **topics: Array of DATA** - (optional) Array of 32 Bytes DATA topics. Topics are order-dependent. Each topic can also be an array of DATA with “or” options.
- **blockhash: DATA, 32 Bytes** - (optional, future) With the addition of EIP-234, blockHash will be a new filter option which restricts the logs returned to the single block with the 32-byte hash blockHash. Using blockHash is equivalent to fromBlock = toBlock = the block number with hash blockHash. If blockHash is present in the filter criteria, then neither fromBlock nor toBlock is allowed.

Returns

- **QUANTITY** - integer of the number of transactions send from this address.

Example

```
curl  https://rpc-endpoint.io:8545 -X POST -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"eth_getLogs","params":[{"topics":
["0x000000000000000000000000a94f5374fce5edbc8e2a8697c15331677e6ebf0b"]}],"id":1}'
```

eth_getCode

Returns code at a given address.

Parameters

- **DATA, 20 Bytes** - address
- **QUANTITY|TAG** - integer block number, or the string "latest"

Returns

- **DATA** - the code from the given address.

Example

```
curl  https://rpc-endpoint.io:8545 -X POST -H "Content-Type: application/json" --data
```

```
'{"jsonrpc":"2.0","method":"eth_getCode","params":["0xa94f5374fce5edbc8e2a8697c15331677e6ebf0b", "0x2"],"id":1}'
```

eth_call

Executes a new message call immediately without creating a transaction on the blockchain.

Parameters

Object - The transaction call object

- **from: DATA, 20 Bytes** - (optional) The address the transaction is sent from.
- **to: DATA, 20 Bytes** - The address the transaction is directed to.
- **gas: QUANTITY** - (optional) Integer of the gas provided for the transaction execution. eth_call consumes zero gas, but this parameter may be needed by some executions.
- **gasPrice: QUANTITY** - (optional) Integer of the gasPrice used for each paid gas
- **value: QUANTITY** - (optional) Integer of the value sent with this transaction
- **data: DATA** - (optional) Hash of the method signature and encoded parameters. For details see Ethereum Contract ABI in the Solidity documentation
- **QUANTITY|TAG** - integer block number, or the string "latest", see the default block paramete

Returns

- **DATA** - the return value of executed contract.

Example

```
curl  https://rpc-endpoint.io:8545 -X POST -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"eth_call","params":[{see above}],"id":1}'
```

eth_getStorageAt

Returns the value from a storage position at a given address.

Parameters

- **DATA, 20 Bytes** - address of the storage.
- **QUANTITY** - integer of the position in the storage.
- **QUANTITY|TAG** - integer block number, or the string "latest"

Returns

- **DATA** - the value at this storage position.

Example

```
curl  https://rpc-endpoint.io:8545 -X POST -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"eth_getStorageAt","params":["0x295a70b2de5e3953354a6a8344e616ed314d7251", "0x0", "latest"],"id":1}'
```

eth_estimateGas

Generates and returns an estimate of how much gas is necessary to allow the transaction to complete. The transaction will not be added to the blockchain. Note that the estimate may be significantly more than the amount of gas actually used by the transaction, for a variety of reasons including EVM mechanics and node performance.

Parameters

Expect that all properties are optional.

Object - The transaction call object

- **from: DATA, 20 Bytes** - The address the transaction is sent from.
- **to: DATA, 20 Bytes** - The address the transaction is directed to.
- **gas: QUANTITY** - Integer of the gas provided for the transaction execution. eth_call consumes zero gas, but this parameter may be needed by some executions.
- **gasPrice: QUANTITY** - Integer of the gasPrice used for each paid gas
- **value: QUANTITY** - Integer of the value sent with this transaction
- **data: DATA** - Hash of the method signature and encoded parameters. For details see Ethereum Contract ABI in the Solidity documentation
- **QUANTITY|TAG** - integer block number, or the string "latest", see the default block paramete

Returns

- **QUANTITY** - the amount of gas used.

Example

```
curl  https://rpc-endpoint.io:8545 -X POST -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"eth_estimateGas","params":[{see above}],"id":1}'
```

eth_newFilter

Creates a filter object, based on filter options. To get all matching logs for specific filter, call eth_getFilterLogs. To check if the state has changed, call eth_getFilterChanges.

Parameters

Object - The filter options:

- **fromBlock: QUANTITY|TAG** - (optional, default: "latest") Integer block number, or "latest" for the last mined block
- **toBlock: QUANTITY|TAG** - (optional, default: "latest") Integer block number, or "latest" for the last mined block
- **address: DATA|Array, 20 Bytes** - (optional) Contract address or a list of addresses from which logs should originate.
- **topics: Array of DATA** - (optional) Array of 32 Bytes DATA topics. Topics are order-dependent. Each topic can also be an array of DATA with “or” options.

Returns

- **QUANTITY** - A filter id.

Example

```
curl  https://rpc-endpoint.io:8545 -X POST -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"eth_newFilter","params":[{"topics":["0x12341234"]}],"id":1}'
```

eth_newBlockFilter

Creates a filter in the node, to notify when a new block arrives. To check if the state has changed, call eth_getFilterChanges.

Parameters

None

Returns

1. QUANTITY - A filter id.

Example

```
curl  https://rpc-endpoint.io:8545 -X POST -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"eth_newBlockFilter","params":[],"id":1}'
```

eth_getFilterLogs

Returns an array of all logs matching filter with given id.

:::caution eth_getLogs vs. eth_getFilterLogs These 2 methods will return the same results for same filter options:

1. eth_getLogs with params [options]
2. eth_newFilter with params [options], getting a [filterId] back, then calling eth_getFilterLogs with [filterId] :::

Parameters

- **QUANTITY** - the filter id.

Returns

Array - Array of log objects, or an empty array

- For filters created with eth_newFilter logs are objects with the following params:
  - **removed: TAG** - true when the log was removed, due to a chain reorganization. false if its a valid log.
  - **logIndex: QUANTITY** - integer of the log index position in the block. null when its pending log.
  - **transactionIndex: QUANTITY** - integer of the transactions index position log was created from. null when its pending log.
  - **transactionHash: DATA, 32 Bytes** - hash of the transactions this log was created from. null when its pending log.
  - **blockHash: DATA, 32 Bytes** - hash of the block where this log was in. null when its pending log.
  - **blockNumber: QUANTITY** - the block number where this log was in. null when its pending log.
  - **address: DATA, 20 Bytes** - address from which this log originated.
  - **data: DATA** - contains one or more 32 Bytes non-indexed arguments of the log.
  - **topics: Array of DATA** - Array of 0 to 4 32 Bytes DATA of indexed log arguments. (In solidity: The first topic is the hash of the signature of the event (e.g. Deposit(address,bytes32,uint256)), except you declared the event with the anonymous specifier.)

Example

```
curl  https://rpc-endpoint.io:8545 -X POST -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"eth_getFilterLogs","params":["0x16"],"id":1}'
```

eth_getFilterChanges

Polling method for a filter, which returns an array of logs that occurred since the last poll.

Parameters

- **QUANTITY** - the filter id.

Returns

Array - Array of log objects, or an empty array if nothing has changed since last poll.

- For filters created with eth_newBlockFilter the return are block hashes (DATA, 32 Bytes), e.g. ["0x3454645634534..."].
- For filters created with eth_newFilter logs are objects with the following params:
  - **removed: TAG** - true when the log was removed, due to a chain reorganization. false if its a valid log.
  - **logIndex: QUANTITY** - integer of the log index position in the block. null when its pending log.
  - **transactionIndex: QUANTITY** - integer of the transactions index position log was created from. null when its pending log.
  - **transactionHash: DATA, 32 Bytes** - hash of the transactions this log was created from. null when its pending log.
  - **blockHash: DATA, 32 Bytes** - hash of the block where this log was in. null when its pending log.
  - **blockNumber: QUANTITY** - the block number where this log was in. null when its pending log.
  - **address: DATA, 20 Bytes** - address from which this log originated.
  - **data: DATA** - contains one or more 32 Bytes non-indexed arguments of the log.
  - **topics: Array of DATA** - Array of 0 to 4 32 Bytes DATA of indexed log arguments. (In solidity: The first topic is the hash of the signature of the event (e.g. Deposit(address,bytes32,uint256)), except you declared the event with the anonymous specifier.)

Example

```
curl  https://rpc-endpoint.io:8545 -X POST -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"eth_getFilterChanges","params":["0x16"],"id":1}'
```

eth_uninstallFilter

Uninstalls a filter with a given id. Should always be called when a watch is no longer needed. Additionally, filters timeout when they aren't requested with eth_getFilterChanges for some time.

Parameters

- **QUANTITY** - The filter id.

Returns

- **Boolean** - true if the filter was successfully uninstalled, otherwise false.

Example

```
curl  https://rpc-endpoint.io:8545 -X POST -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"eth_uninstallFilter","params":["0xb"],"id":1}'
```

eth_unsubscribe

Subscriptions are cancelled with a regular RPC call with eth_unsubscribe as a method and the subscription id as the first parameter. It returns a bool indicating if the subscription was cancelled successfully.

Parameters

- **SUBSCRIPTION ID**

Returns

- **UNSUBSCRIBED FLAG** - true if the subscription was cancelled successful.

Example

```
curl  https://rpc-endpoint.io:8545 -X POST -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"eth_unsubscribe","params":["0x9cef478923ff08bf67fde6c64013158d"],"id":1}'
```

---

id: json-rpc-net title: Net description: "List of Net JSON RPC commands for Polygon Edge." keywords: - docs - polygon - edge - json - rpc - commands

import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem'; import {useState} from 'react';

net_version

Returns the current network id.

Parameters

None

Returns

- **String** - The current network id.

Example

```
curl  https://rpc-endpoint.io:8545 -X POST -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"net_version","params":[],"id":83}'
```

net_listening

Returns true if a client is actively listening for network connections.

Parameters

None

Returns

- **Boolean** - true when listening, otherwise false.

Example

```
curl  https://rpc-endpoint.io:8545 -X POST -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"net_listening","params":[],"id":83}'
```

net_peerCount

Returns number of peers currently connected to the client.

Parameters

None

Returns

- **QUANTITY** - integer of the number of connected peers.

Example

```
curl  https://rpc-endpoint.io:8545 -X POST -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"net_peerCount","params":[],"id":1}'
```

---

id: json-rpc-txpool title: TxPool description: "List of TxPool JSON RPC commands for Polygon Edge." keywords: - docs - polygon - edge - json - rpc - commands

import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem'; import {useState} from 'react';

txpool_content

Returns a list with the exact details of all the transactions currently pending for inclusion in the next block(s), as well as the ones that are being scheduled for future execution only.

Parameters

None

Example

```
curl  https://rpc-endpoint.io:8545 -X POST -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"txpool_content","params":[],"id":1}'
```

txpool_inspect

Returns a list with a textual summary of all the transactions currently pending for inclusion in the next block(s), as well as the ones that are being scheduled for future execution only. This is a method specifically tailored to developers to quickly see the transactions in the pool and find any potential issues.

Parameters

None

Example

```
curl  https://rpc-endpoint.io:8545 -X POST -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"txpool_inspect","params":[],"id":1}'
```

txpool_status

Returns the number of transactions currently pending for inclusion in the next block(s), as well as the ones that are being scheduled for future execution only.

Parameters

None

Example

```
curl  https://rpc-endpoint.io:8545 -X POST -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"txpool_status","params":[],"id":1}'
```

---

id: json-rpc-web3 title: Web3 description: "List of Web3 JSON RPC commands for Polygon Edge." keywords: - docs - polygon - edge - json - rpc - commands

import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem'; import {useState} from 'react';

web3_clientVersion

Returns the current client version.

Parameters

None

Returns

- **String** - The current client version

Example

```
curl  https://rpc-endpoint.io:8545 -X POST -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"web3_clientVersion","params":[],"id":1}'
```

web3_sha3

Returns Keccak-256 (not the standardized SHA3-256) of the given data.

Parameters

- **DATA** - the data to convert into a SHA3 hash

Returns

- **DATA** - The SHA3 result of the given string.

Example

```
curl  https://rpc-endpoint.io:8545 -X POST -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"web3_sha3","params":["0x68656c6c6f20776f726c64"],"id":1}'
```

---

id: checkpoint-manager title: CheckpointManager description: "The Checkpoint Manager contract for Polygon Edge" keywords: - docs - polygon - edge - core - checkpoint - manager

Overview

The CheckpointManager contract is used by validators to submit signed checkpoints as proof of the canonical chain. The contract verifies that the provided signature is valid and that the checkpoint has been signed as expected. The contract also includes an initialization function that can only be called once to set the contract's dependencies and domain.

Functions

initialize

```
function initialize(
    IBLS newBls,
    IBN256G2 newBn256G2,
    bytes32 newDomain,
    uint256 chainId_,
    Validator[] calldata newValidatorSet
 ) external initializer {
    // Implementation details ...
}
```

This function is the initializer for the CheckpointManager contract. It sets the following contract dependencies and parameters:

- `newBls` : The address of the BLS library contract.

- `newBn256G2` : The address of the BN256G2 library contract.
- `newDomain` : The domain to use when hashing messages to a point.
- `chainId_` : The chain ID of the Ethereum network.
- `newValidatorSet` : The array of validators to seed the contract with. This is passed as calldata so that the contract doesn't have to be initialized with a static length.

submit

```
  function submit(
    CheckpointMetadata calldata checkpointMetadata,
    Checkpoint calldata checkpoint,
    uint256[2] calldata signature,
    Validator[] calldata newValidatorSet,
    bytes calldata bitmap
  ) external {
      // Implementation details ...
}
```

This function is used by validators to submit a new checkpoint.

- `checkpointMetadata` : Contains metadata about the submitted checkpoint, such as the current validator set hash and block hash.
- `checkpoint` : The checkpoint data, including the block number, epoch, event root, and nonce.
- `signature` : The signature from a group of validators, verifying the submitted checkpoint data.
- `newValidatorSet` : An array of updated validators for the contract.
- `bitmap` : The bitmap of the signatures, indicating which validators signed the checkpoint.

getEventMembershipByBlockNumber

```
function getEventMembershipByBlockNumber(
      uint256 blockNumber,
      bytes32 leaf,
      uint256 leafIndex,
      bytes32[] calldata proof
  ) external view returns (bool) {
    // Implementation details ...
  }
```

This function returns a boolean indicating whether the specified leaf is a member of the event root corresponding to the given blockNumber.

- `blockNumber` : The block number to retrieve the event root for.
- `leaf` : The leaf to check membership for.
- `leafIndex` : The index of the leaf in the Merkle tree.
- `proof` : The Merkle proof demonstrating membership of the leaf in the Merkle tree.

getEventMembershipByEpoch

```
function getEventMembershipByEpoch(
      uint256 epoch,
      bytes32 leaf,
      uint256 leafIndex,
      bytes32[] calldata proof
  ) external view returns (bool) {
    // Implementation details ...
  }
```

This function returns the membership status of an event (specified by `leaf` and `leafIndex`) in an event merkle tree (specified by `eventRoot`) during a certain epoch of the Ethereum network.

- `epoch` : The epoch for which to check event membership.
- `leaf` : The 32-byte hash of the event.
- `leafIndex` : The position of the event in the event merkle tree.
- `proof` : The merkle proof to validate the membership of the event in the tree.
- The function returns true if the event is a member of the tree, and false otherwise.

getEventRootByBlock

```
  function getEventRootByBlock(uint256 blockNumber) external view returns (bytes32 root) {
      // Implementation details ...
  }
```

This function returns the Merkle root of the event log for a specific block number.

- `blockNumber` : The number of the block for which to retrieve the checkpoint Merkle tree root.
- `root` : The root of the Merkle tree of checkpoints for the specified block.

_setNewValidatorSet

:::caution Current validator set

To bootstrap the network, the root validator is composed of internal validators of Polygon Labs.

:::

```
function _setNewValidatorSet(Validator[] calldata newValidatorSet) private {
    // Implementation details ...
  }
```

This is an internal function used to set a new validator set for the contract.

_verifySignature

```
function _verifySignature(
        uint256[2] memory message,
        uint256[2] calldata signature,
        bytes calldata bitmap
    ) private view {
    // Implementation details ...
  }
```

This function checks the validity of a checkpoint by verifying the signature included in the checkpoint against a computed aggregate public key of the validators and a message (both passed as inputs).

- `message` : A 2-element array representing the message to verify.
- `signature` : A 2-element array representing the signature to verify against the message and aggregate public key.
- `bitmap` : A byte array representing the bitmap indicating which validators were involved in signing the message.
- The function performs the following checks and throws an error if any check fails:
  - Bitmap is not empty
  - Total voting power represented by validators in the bitmap is greater than 2/3 of the total voting power of all validators
  - The signature is successfully verified against the message and aggregate public key.

verifyCheckpoint

```
function _verifyCheckpoint(uint256 prevId, Checkpoint calldata checkpoint) private view {
    Checkpoint memory oldCheckpoint = checkpoints[prevId];
    require(
    // Implementation details ...
    );
}
```

This function verifies a checkpoint by checking its validity.

- `prevId` : The ID of the current checkpoint.
- `checkpoint` : The checkpoint to be verified.

The function checks if the checkpoint has a valid epoch value, either equal to the epoch value of the current checkpoint or one greater than it. It also checks if the `blockNumber` of the checkpoint is greater than the `blockNumber` of the current checkpoint. If both checks pass, the function returns true, otherwise it throws an error.

_getValueFromBitmap

```
function _getValueFromBitmap(bytes calldata bitmap, uint256 index) private pure returns (bool) {
    // Implementation details ...
}
```

This is an internal function which takes a `bitmap` and an `index` as inputs and returns a boolean value. It works as follows:

- `byteNumber` is calculated as index / 8
- `bitNumber` is calculated as index % 8
- If `byteNumber` is greater than or equal to the length of `bitmap` , it returns false.
- It returns the result of the expression uint8(bitmap[byteNumber]) & (1 << `bitNumber`) > 0, which checks if the bitNumber-th bit of the byteNumber-th byte in bitmap is set to 1.

---

id: exit-helper title: ExitHelper description: "The ExitHelper contract for Polygon Edge" keywords: - docs - polygon - edge - core - exit - exit helper

Overview

The ExitHelper contract is a smart contract that allows for the process of exits, when a user wants to withdraw their assets from a `childchain` to a `rootchain`. It uses a Merkle tree to check whether a state transition is included in a block and provides functions to process exits and batch exits. The contract also has an event trigger that is emitted when an exit is processed. The contract is initialized with a `CheckpointManager` address.

Mappings

The following mappings are used in the contract:

- `processedExits`: a mapping from uint256 (exit id) to bool indicating whether an exit with a particular `id` has already been processed

The `processedExits` mapping is used to track which exits have already been processed and prevent duplicate processing of the same exit.

Events

ExitProcessed

```
event ExitProcessed(uint256 indexed id, bool indexed success, bytes returnData);
```

The contract includes an event called `ExitProcessed`, which is emitted when an exit is processed. The event includes the following indexed fields:

- `id`: the identifier of the exit
- `success`: the result of processing the exit
- `returnData`: return data from the onL2StateReceive function

Functions

initialize

```
function initialize(ICheckpointManager newCheckpointManager) external initializer {
    require(
        address(newCheckpointManager) != address(0) && address(newCheckpointManager).code.length != 0,
        "ExitHelper: INVALID_ADDRESS"
    );
    checkpointManager = newCheckpointManager;
}
```

This function is used to initialize the contract with the address of the CheckpointManager contract. It takes the following parameter:

- `newCheckpointManager`: the address of the checkpoint manager contract

The function requires that the address of the checkpoint manager contract is not the zero address and that the contract has deployed code. If either of these checks fail, the contract will revert with an appropriate error message.

exit

```
function exit(
    uint256 blockNumber,
    uint256 leafIndex,
    bytes calldata unhashedLeaf,
    bytes32[] calldata proof
) external onlyInitialized {
    // Implementation details ...
}
```

This function allows anyone to initiate an exit by calling the function and passing in the block `number`, leaf `index`, unhashed `leaf`, and `proof`. The function will check that the contract has been initialized by calling the `onlyInitialized` modifier, which checks that the `checkpointManager` address is not the zero address.

The function will then call the `_exit` function and pass the parameters to it.

batchExit

```
function batchExit(BatchExitInput[] calldata inputs) external onlyInitialized{
    // Implementation details ...
}
```

This function allows anyone to initiate multiple exits at once by calling the function and passing in an array of `BatchExitInput` structs. The function will check that the contract has been initialized by calling the `onlyInitialized` modifier, which checks that the `checkpointManager` address is not the zero address.

The function will then loop through the inputs array, calling the `_exit` function for each input.

_exit

```
function _exit(
    uint256 blockNumber,
```

```
    uint256 leafIndex,
    bytes calldata unhashedLeaf,
    bytes32[] calldata proof,
    bool isBatch
) private{
   // Implementation details ...
}
```

This function is the internal function that handles the actual execution of an `exit`. It takes the following parameters:

- `blockNumber` : the block number of the state
- `leafIndex` : the leaf index of the state in the Merkle tree
- `unhashedLeaf` : the unhashed leaf of the state
- `proof` : the Merkle proof of the state
- `isBatch` : a boolean indicating whether the exit is part of a batch or not

This function first checks the `processedExits` mapping to make sure that the `exit` with the given `id` has not been processed before.

Then it calls the checkpointManager's `getEventMembershipByBlockNumber` function and passing `blockNumber`, `keccak256(unhashedLeaf)`, `leafIndex`, `proof` as arguments, this function verifies the proof of inclusion of the unhashed leaf in the Merkle tree of the given block number. If the proof is valid, it marks the exit as processed and calls the `onL2StateReceive` function of the receiver contract with the `id`, `sender` and data of the `exit` and emits the `ExitProcessed` event to indicate whether the call was successful or not.

The `isBatch` parameter is used to determine if this is a single exit or part of a batch of exits, if it's a batch, the function will not revert and just return if the exit has been processed before.

id: reward-pool title: RewardPool description: "The smart contract for the reward mechanism for Edge" keywords: - docs - polygon - edge - reward - rewards - pool

Overview

`RewardPool` is library for the management and distributiion validator rewards. Each validator has a reward pool for delegators delegating funds to them.

Functions

distributeReward

```
function distributeReward(RewardPool storage pool, uint256 amount) internal {
    if (amount == 0) return;
    if (pool.supply == 0) revert NoTokensDelegated(pool.validator);
    pool.magnifiedRewardPerShare += (amount * magnitude()) / pool.supply;
}
```

This function distributes an amount of rewards to the `pool`. If amount is 0 or `pool.supply` is 0, this function does nothing. Otherwise, `pool.magnifiedRewardPerShare` is incremented by amount multiplied by `magnitude` divided by `pool.supply`.

deposit

```
function deposit(
    RewardPool storage pool,
    address account,
    uint256 amount
) internal {
    pool.balances[account] += amount;
    pool.supply += amount;
    pool.magnifiedRewardCorrections[account] -= (pool.magnifiedRewardPerShare * amount).toInt256Safe();
}
```

This function credits the balance of account in the `pool` by amount. `pool.balances[account]` is incremented by amount, `pool.supply` is incremented by amount, and `pool.magnifiedRewardCorrections[account]` is decremented by amount multiplied by `pool.magnifiedRewardPerShare` after being converted to an int256 using `toInt256Safe`.

withdraw

```
function withdraw(
    RewardPool storage pool,
    address account,
    uint256 amount
) internal {
    pool.balances[account] -= amount;
    pool.supply -= amount;
```

```
    pool.magnifiedRewardCorrections[account] += (pool.magnifiedRewardPerShare * amount).toInt256Safe();
}
```

This function decrements the balance of account in the `pool` by amount. `pool.balances[account]` is decremented by amount, `pool.supply` is decremented by amount, and `pool.magnifiedRewardCorrections[account]` is incremented by amount multiplied by `pool.magnifiedRewardPerShare` after being converted to an int256 using `toInt256Safe()` .

claimRewards

```
function claimRewards(RewardPool storage pool, address account) internal returns (uint256 reward) {
    reward = claimableRewards(pool, account);
    pool.claimedRewards[account] += reward;
}
```

This function increments `pool.claimedRewards[account]` by the claimable rewards for account in the `pool` , and returns the amount of rewards claimed. The claimable rewards for account can be calculated using `claimableRewards` .

balanceOf

```
function balanceOf(RewardPool storage pool, address account) internal view returns (uint256) {
    return pool.balances[account];
}
```

This function returns the `balance` (i.e. amount delegated) of account in the `pool` .

totalRewardsEarned

```
function totalRewardsEarned(RewardPool storage pool, address account) internal view returns (uint256) {
    int256 magnifiedRewards = (pool.magnifiedRewardPerShare * pool.balances[account]).toInt256Safe();
    uint256 correctedRewards = (magnifiedRewards + pool.magnifiedRewardCorrections[account]).toUint256Safe();
    return correctedRewards / magnitude();
}
```

This function returns the total rewards earned by account in the `pool` . This is calculated by first multiplying `pool.magnifiedRewardPerShare` by `pool.balances[account]` , converting the result to an int256 using `toInt256Safe` , and then adding `pool.magnifiedRewardCorrections[account]` . The final result is then divided by `magnitude` and returned as a `uint256` .

claimableRewards

```
function claimableRewards(RewardPool storage pool, address account) internal view returns (uint256) {
    return totalRewardsEarned(pool, account).sub(pool.claimedRewards[account]);
}
```

This function returns the amount of claimable rewards for account in the pool. This is calculated by subtracting `pool.claimedRewards[account]` from `totalRewardsEarned` .

magnitude

```
function magnitude() private pure returns (uint256) {
        return 1e18;
    }
```

This function returns the `magnitude` of the reward multiplier.

id: state-receiver title: StateReceiver description: "The State receiver contract for Polygon Edge" keywords: - docs - polygon - edge - core - checkpoint - manager

Overview

The StateReceiver contract is a smart contract that allows for the execution and relay of state data on a child blockchain. It uses a Merkle tree to commit to a batch of state sync objects and provides functions to verify the inclusion of individual state sync objects in a commitment. It also allows for the execution of multiple state sync events at once and has a mapping to keep track of processed state sync events. The contract also has several event triggers that are emitted when specific actions are taken, such as when a new commitment is added, when a state sync event is executed, and when the execution of a state sync event fails.

Structs

StateSync

```
struct StateSync {
    uint256 id;
    address sender;
    address receiver;
    bytes data;
}
```

The `StateSync` struct is used to represent a single state sync event. It includes the following fields:

- `id` : a unique identifier for the state sync event
- `sender` : the address of the sender of the state sync event
- `receiver` : the address of the receiver of the state sync event
- `data` : the data associated with the state sync event

StateSyncCommitment

```
struct StateSyncCommitment {
        uint256 startId;
        uint256 endId;
        bytes32 root;
    }
```

The `StateSyncCommitment` struct is used to represent a group of state sync events that are processed together. It includes the following fields:

- `startId` : the identifier of the first state sync event in the bundle
- `endId` : the identifier of the last state sync event in the bundle
- `root` : the root of the Merkle tree formed by the state sync events in
- the bundle

Mappings

The following mappings are used in the contract:

- `processedStateSyncs` : a mapping from uint256 (state sync id) to bool indicating whether a state sync event with a particular id has already been processed
- `commitments` : a mapping from uint256 (commitment id) to StateSyncCommitment struct indicating the commitment associated with a particular state sync id
- `commitmentIds` : an array of uint256 representing the ids of all commitments in the contract

The `processedStateSyncs` mapping is used to track which state sync events have already been executed and prevent duplicate execution of the same event. The `commitments` mapping is used to store `commitments` and retrieve them later by state sync id. `commitmentIds` is used to keep track of the order of `commitments` in the contract.

Constants

```
uint256 private constant MAX_GAS = 300000;
```

The contract defines a constant called `MAX_GAS` , which represents the maximum gas allowed for each message call.

Events

StateSyncResult

```
event StateSyncResult(uint256 indexed counter, ResultStatus indexed status, bytes32 message);
```

The contract includes an event called `StateSyncResult` , which is emitted when a `StateSync` event is processed. The event includes the following indexed fields:

- `counter` : the identifier of the state sync event
- `status` : the result of processing the state sync event (success, failure, or skip)
- `message` : a message associated with the result of processing the state sync event

NewCommitment

```
event NewCommitment(uint256 indexed startId, uint256 indexed endId, bytes32 root);
```

The contract also includes an event called `NewCommitment` , which is emitted when a new commitment is made. The event includes the following indexed fields:

- `startId` : the startId of the commitment
- `endId` : the endId of the commitment
- `root` : the root of the Merkle tree of the commitment

Functions

commit

```
function commit(
    StateSyncCommitment calldata commitment,
    bytes calldata signature,
    bytes calldata bitmap
) external onlySystemCall
```

This function is used to commit the root of a Merkle tree formed by a group of state sync events. It takes the following parameters:

- `commitment` : a StateSyncCommitment struct representing the group of state sync events to be committed
- `signature` : the signature of the commitment
- `bitmap` : a bitmap indicating which validators signed the commitment

The function checks that the `startId` of the `commitment` is the `lastCommittedId` plus 1, and that the `endId` of the `commitment` is greater than or equal to the `startId` . It then verifies the `signature` of the `commitment` using an aggregation of public keys. If the `signature` is valid, it stores the `commitment` in the `commitments` mapping and updates the `lastCommittedId` .

execute

```
function execute(bytes32[] calldata proof, StateSync[] calldata objs) external
```

This function is used to submit a leaf of a Merkle tree formed by a group of state sync events for execution. It takes the following parameters:

- `proof` : an array of Merkle proofs for the state sync events in the group
- `objs` : an array of StateSync structs representing the state sync events in the group

The function first checks that there is a `bundle` in the `bundles` mapping that has not yet been executed. It then calculates the data hash of the objs array and verifies that the data hash is a leaf in the Merkle tree rooted at the root of the corresponding `StateSyncBundle` struct. If the data hash is a valid leaf, the function increments the counter and updates the `currentLeafIndex` and `lastExecutedBundleCounter` as necessary. Finally, it processes each state sync event in the objs array by calling the processSync function for each event.

batchExecute

```
function batchExecute(bytes32[][] calldata proofs, StateSync[] calldata objs) external
```

This function is similar to the execute function but allows for multiple state sync events to be executed at once. It takes the following parameters:

- `proofs` : an array of arrays of Merkle proofs for the state sync events in the group
- `objs` : an array of StateSync structs representing the state sync events in the group

The function first checks that the length of `proofs` is equal to the length of `objs` . It then checks that there is a commitment in the commitments mapping that has not yet been executed for each element in `objs` . It then calculates the data hash of each `obj` and verifies that the data hash is a leaf in the Merkle tree rooted at the root of the corresponding commitment. If the proof is valid, it executes the corresponding state sync event.

getRootByStateSyncId

```
function getRootByStateSyncId(uint256 id) external view returns (bytes32)
```

This function allows for retrieval of the root data for a specific state sync event. It takes the following parameter:

- `id` : the id of the state sync event

It first retrieves the root from the commitment with the highest `id` lower than or equal to the provided `id` . It then checks if the root is not equal to `0` and returns it if it is not, otherwise it throws an error.

getCommitmentByStateSyncId

```
function getCommitmentByStateSyncId(uint256 id) public view returns (StateSyncCommitment memory)
```

This function allows to retrieve the commitment for a state sync `id` .

It takes the following parameter:

- `id` : the state sync id to get the commitment for

The function first retrieves the index of the commitment in `commitmentIds` using the `findUpperBound` function, passing the given `id` . It then checks if the index is not equal to the length of `commitmentIds` , if not, it returns the commitment from the `commitments` mapping using the calculated `index` . If it is, it will throw an error `StateReceiver: NO_COMMITMENT_FOR_ID`

_executeStateSync

```
function _executeStateSync(StateSync calldata obj) private
```

This function is an internal function to execute a state sync object.

It takes the following parameter:

- `obj` : the `StateSync` object to be executed

The function first checks if the state sync has already been processed by checking the `processedStateSyncs` mapping.

If it has, it will throw an error `StateReceiver: STATE_SYNC_IS_PROCESSED` . If not, it will check if the receiver address has no code, if so, it will emit a `StateSyncResult` event with false status, otherwise, it will mark the state sync as processed and call the receiver address with the onStateReceive function and the obj `id` , sender and data as parameters. It will then emit a `StateSyncResult` event with the success status and return data of the call.

_checkPubkeyAggregation

```
function _checkPubkeyAggregation(bytes32 message, bytes calldata signature, bytes calldata bitmap) internal view
```

This function verifies an aggregated BLS signature using BLS precompile. It takes the following parameters:

- `message` : plaintext of signed message
- `signature` : the signed message
- `bitmap` : bitmap of which validators have signed

The function uses the V `ALIDATOR_PKCHECK_PRECOMPILE` address to call the `staticcall` function with the `abi.encode` of the message, `signature` and `bitmap` passed as parameters. It then decodes the return data to a bool and checks if the call was successful and the signature is verified. If not it will throw an error `SIGNATURE_VERIFICATION_FAILED` .

id: state-sender title: StateSender description: "The State Sender contract for Polygon Edge" keywords: - docs - polygon - edge - core - checkpoint - manager

Overview

The `StateSender` contract is a simple smart contract that sends messages to a child. Messages are indexed by validators from root and then signed. Once they have enough signatures, they can be committed on `StateReceiver` on child.

Unlike the current implementation of child, sending and receiving messages is split into two contracts on root, where this one sends the state.

Functions

stateSync

```
function syncState(address receiver, bytes calldata data) external {
        // check receiver
        require(receiver != address(0), "INVALID_RECEIVER");
        // check data length
        require(data.length <= MAX_LENGTH, "EXCEEDS_MAX_LENGTH");

        // State sync id will start with 1
        emit StateSynced(++counter, msg.sender, receiver, data);
    }
```

This function allows anyone to emit a `StateSynced` event by calling the function and passing in a receiver `address` and `data` . The function has two require statements: one that checks that the receiver address is not the zero address, and one that checks that the length of the data is less than or equal to the `MAX_LENGTH` , which is **2048**. If either of these checks fail, the contract will revert with an appropriate error message.

If the checks pass, the function will emit a `StateSynced` event with the `id` , `sender` , `receiver` , and `data` .

id: validator-queue title: ValidatorQueue description: "The validator queue contract for Edge." keywords: - docs - polygon - edge - core - validator - queue

Overview

`ValidatorQueue` is a library that can be used to manage a queue of updates to block validators. The queue is used to register new validators, add or remove stake, delegate or undelegate stake.

The queue is processed and cleared at the end of each epoch.

Functions

insert

```
function insert(
    ValidatorQueue storage self,
    address validator,
    int256 stake,
    int256 delegation
) internal {
    uint256 index = self.indices[validator];
    if (index == 0) {
        // insert into queue
        // use index starting with 1, 0 is empty by default for easier checking of pending balances
        index = self.queue.length + 1;
        self.indices[validator] = index;
        self.queue.push(QueuedValidator(validator, stake, delegation));
    } else {
        // update values
        QueuedValidator storage queuedValidator = self.queue[indexOf(self, validator)];
        queuedValidator.stake += stake;
        queuedValidator.delegation += delegation;
    }
}
```

This function is used to queue a validator's `data`. It takes three arguments: the `ValidatorQueue` struct, the `address` of the validator and the change to the validator's `stake` and `delegation`. If the validator is not already in the queue, the validator is added to the queue by adding a new `QueuedValidator` to the queue array and updating the mapping of validator addresses to their position in the queue.

If the validator is already in the queue, the function updates the values of the validator's `stake` and `delegation` in the `QueuedValidator` struct.

resetIndex

```
function resetIndex(ValidatorQueue storage self, address validator) internal {
    self.indices[validator] = 0;
}
```

This function is used to delete `data` from a specific validator in the `queue`. It takes two arguments: the `ValidatorQueue` struct and the `address` of the validator to remove the `queue` data of.

The function updates the mapping of validator addresses to their position in the `queue` by setting the index for the validator to 0, indicating that the validator is not in the `queue`.

reset

```
function reset(ValidatorQueue storage self) internal {
    delete self.queue;
}
```

This function is used to reinitialize the validator `queue`. It takes one argument: the `ValidatorQueue` struct.

The function deletes the `queue` array, effectively resetting the `queue`.

get

```
function get(ValidatorQueue storage self) internal view returns (QueuedValidator[] storage) {
    return self.queue;
}
```

This function is used to return the `queue`. It takes one argument: the `ValidatorQueue` struct.

waiting

```
function waiting(ValidatorQueue storage self, address validator) internal view returns (bool) {
    return self.indices[validator] != 0;
}
```

This function is used to check if a specific validator is in the `queue`. It takes two arguments: the `ValidatorQueue` struct and the `address` of the validator to check.

pendingState

```
function pendingStake(ValidatorQueue storage self, address validator) internal view returns (int256) {
    return self.queue[self.indices[validator]].stake;
}
```

This function is used to return the change of `stake` for a validator in the `queue`. It takes two arguments: the `ValidatorQueue` struct and the `address` of the validator to check the change to `stake` of.

pendingDelegation

```
function pendingDelegation(ValidatorQueue storage self, address validator) internal view returns (int256) {
    return self.queue[self.indices[validator]].delegation;
}
```

This function is used to return the change to `delegation` for a validator in the `queue`. It takes two arguments: the `ValidatorQueue` struct and the `address` of the validator to check the change to delegation of.

id: validator-set title: RootValidatorSet description: "The root validator set contract for Polygon Edge." keywords: - docs - polygon - edge - core - checkpoint - manager

Overview

The `RootValidatorSet` contract is an upgradeable contract that is used to register validators and their ECDSA and BLS public keys on Polygon.

:::caution Current validator set

To bootstrap, the `RootValidatorSet` is composed of internal validators of Polygon Technology.

:::

Functions

initialize

```
function initialize(
    address governance,
    address newCheckpointManager,
    address[] calldata validatorAddresses,
    uint256[4][] calldata validatorPubkeys
) external initializer {
    require(validatorAddresses.length == validatorPubkeys.length, "LENGTH_MISMATCH");
    // slither-disable-next-line missing-zero-check
    checkpointManager = newCheckpointManager;
    uint256 currentId = 0; // set counter to 0 assuming validatorId is currently at 0 which it should be...
    for (uint256 i = 0; i < validatorAddresses.length; i++) {
        Validator storage newValidator = validators[++currentId];
        newValidator._address = validatorAddresses[i];
        newValidator.blsKey = validatorPubkeys[i];

        validatorIdByAddress[validatorAddresses[i]] = currentId;

        emit NewValidator(currentId, validatorAddresses[i], validatorPubkeys[i]);
    }
    currentValidatorId = currentId;
    _transferOwnership(governance);
}
```

This function is the initialization function for the `RootValidatorSet` contract. It is used to set up the initial state of the contract. The function takes in four arguments:

- `governance`: the address of the governance contract that will own the `RootValidatorSet` contract after it is initialized
- `newCheckpointManager`: the address of the `CheckpointManager` for the contract
- `validatorAddresses`: an array of validator addresses to seed the contract with
- `validatorPubkeys`: an array of validator pubkeys to seed the contract with

The function has a require statement that checks that the length of `validatorAddresses` is equal to the length of the `validatorPubkeys`. If these are not the same length, the function reverts with an error message.

The function then sets the value of the `checkpointManager` to the `newCheckpointManager` argument. It then iterates over `validatorAddresses` and registers each address as a validator. It does this by creating a new `Validator` and storing it in the `validators` mapping using the next available validator `id` as the `key`. The `validatorIdByAddress` mapping is also updated to store the validator `id` corresponding to the validator `address`.

After all the validators have been registered, `currentValidatorId` is updated to the highest validator `id` that was assigned.

Finally, contract ownership is transferred to the `governance` contract that was passed in as an argument.

addValidators

```
function addValidators(Validator[] calldata newValidators) external {
    require(msg.sender == checkpointManager, "ONLY_CHECKPOINT_MANAGER");
    uint256 length = newValidators.length;
    uint256 currentId = currentValidatorId;

    for (uint256 i = 0; i < length; i++) {
        validators[i + currentId + 1] = newValidators[i];

        emit NewValidator(currentId + i + 1, newValidators[i]._address, newValidators[i].blsKey);
    }

    currentValidatorId += length;
}
```

This function allows the `checkpointManager` to add new validators to the contract. It takes in an array of Validators and iterates over them, adding each one to the contract and emitting a `NewValidator` event for each one.

The function also increments `currentValidatorId` by the length of the array. The function has a require statement that checks that the caller is the `checkpointManager`, and will revert with an error message if the caller is not.

getValidators

```
function getValidator(uint256 id) external view returns (Validator memory) {
    return validators[id];
}
```

This function allows anyone to retrieve the `Validator` struct for a specific validator by `id`. It takes in an `id` as an argument and returns the corresponding `Validator` struct from the `validators` mapping.

getValidatorBlsKey

```
function getValidatorBlsKey(uint256 id) external view returns (uint256[4] memory) {
    return validators[id].blsKey;
}
```

This function is a convenience function that allows anyone to retrieve the BLS public key of a specific validator by `id`. It takes in an `id` as an argument and returns the `blsKey` stored in the `Validator` struct for that `id`.

getValidatorId

```
function getValidatorId(address address) external view returns (uint256) {
    return validatorIdByAddress[address];
}
```

This function allows anyone to retrieve the `id` of a validator by their `address`. It takes in an `address` as an argument and returns the corresponding `id` from the `validatorIdByAddress` mapping.

getValidatorCount

```
function getValidatorCount() external view returns (uint256) {
    return currentValid;
}
```

This function allows anyone to retrieve the total number of validators that are registered in the contract. It returns the value of `currentValidatorId`, which is the highest validator `id` that has been assigned to a validator. The `currentValidatorId` is incremented each time a new validator is added to the contract, so the number returned by this function represents the total number of validators that have been registered in the contract.

isValidator

```
function isValidator(address _address) external view returns (bool) {
    return validatorIdByAddress[_address] != 0;
}
```

This function allows anyone to check if a given address is registered as a validator. It takes in an address as an argument and indicates whether the address is registered as a validator.

The function checks the `validatorIdByAddress` mapping to see if the given `address` has an `id` associated with it.

isValidatorActive

```
function isValidatorActive(uint256 _validatorId) external view returns (bool) {
    return _validatorId <= currentValidatorId && _validatorId > currentValidatorId - ACTIVE_VALIDATOR_SET_SIZE;
}
```

This function allows anyone to check if a given validator is active. It takes in a validator `id` as an argument and indicates whether the validator is active.

The function checks if the given `id` is within the range of `ids` that correspond to the active validator set. The active validator set is defined as the most recent `ACTIVE_VALIDATOR_SET_SIZE` number of validators.

isCallerValidator

```
function isCallerValidator() external view returns (bool) {
    return isValidator(msg.sender);
}
```

This function is a convenience function that allows anyone to check if the caller of the function is a validator. It indicates whether the caller is a validator. The function does this by calling `isValidator` and passing in the caller's `address` as an argument.

isCallerActiveValidator

```
function isCallerActiveValidator() external view returns (bool) {
    return isValidatorActive(getValidatorId(msg.sender));
}
```

This function is a convenience function that allows anyone to check if the caller of the function is an active validator. It indicates whether the caller is an active validator. The function does this by calling `isValidatorActive` and passing in the caller's validator `id` as an argument. The validator `id` is retrieved by calling `getValidatorId` and passing in the caller's address as an argument.

id: withdrawal-queue title: WithdrawalQueue description: "The withdrawal queue logic for Polygon Edge." keywords: - docs - polygon - edge - core - withdrawal - queue

Overview

The `WithdrawalQueueLib` library provides functions for managing a queue of withdrawals. The library defines two data types: `Withdrawal` and `WithdrawalQueue`.

Data Types

Withdrawal

The `Withdrawal` struct represents a withdrawal from the contract. It has the following fields:

- `amount` : The amount to withdraw.
- `epoch` : The epoch of the withdrawal.

WithdrawalQueue

The `WithdrawalQueue` struct represents a queue of Withdrawal structs. It has the following fields:

- `head` : The earliest unprocessed index (the most recently filled withdrawal).
- `tail` : The index of the most recent withdrawal (the total number of submitted withdrawals).
- `withdrawals` : A mapping from index to Withdrawal struct, representing the withdrawals in the queue.

Functions

append

```
function append(
    WithdrawalQueue storage self,
    uint256 amount,
    uint256 epoch
) internal
```

This function updates the `WithdrawalQueue` with new withdrawal data. If there is already a withdrawal for the epoch being submitted, the `amount` will be added to that `epoch` ; otherwise, a new `Withdrawal` struct will be created in the queue.

The function takes the following arguments:

- `self` : The WithdrawalQueue struct represents the queue of withdrawals.
- `amount` : The amount to withdraw.
- `epoch` : The epoch of the withdrawal.

withdrawable

```
function withdrawable(
    WithdrawalQueue storage self,
    uint256 currentEpoch
) internal view returns (uint256 amount, uint256 newHead)
```

This function returns the amount withdrawable through a specified epoch and the new head index. It is meant to be used with the current epoch being passed in.

The function takes the following arguments:

- `self` : The WithdrawalQueue struct.
- `currentEpoch` : The epoch to check from.

The function returns a tuple containing the following:

- `amount` : The amount withdrawable through the specified epoch.
- `newHead` : The head of the queue once these withdrawals have been processed.

pending

```
function pending(
    WithdrawalQueue storage self,
    uint256 currentEpoch
) internal view returns (uint256 amount)
```

This function returns the amount withdrawable beyond a specified `epoch` . It is meant to be used with the current epoch being passed in.

The function takes the following arguments:

- `self` : The WithdrawalQueue struct.
- `currentEpoch` : The epoch to check from.

The function returns the `amount` withdrawable from beyond the specified `epoch` .

pop

```
function pop(WithdrawalQueue storage self) internal
```

This function removes the earliest unprocessed withdrawal from the `WithdrawalQueue` . It updates the head field of the `WithdrawalQueue` struct to reflect this change.

The function takes the following argument:

- `self` : The `WithdrawalQueue` struct.

clear

```
function clear(WithdrawalQueue storage self) internal
```

This function removes all withdrawals from the `WithdrawalQueue` . It resets the `head` and `tail` fields of the `WithdrawalQueue` struct to `0` .

The function takes the following argument:

- self: The `WithdrawalQueue` struct.

---

id: supernets-blockchain title: Blockchain Mechanics sidebar_label: Blockchain description: "The state database for Supernets." keywords: - docs - polygon - edge - stake - unstake - validator - epoch

Supernets are built on a common blockchain design that effectively manages and maintains the blockchain data structure, consisting of a sequential chain of blocks containing transactions and other metadata; a state database.

The core blockchain implementation offers various functionalities, such as:

- Adding new blocks to the chain.
- Retrieving blocks using their hash or number.
- Handling chain reorganizations (i.e., switching to a different chain with greater difficulty).
- Verifying block headers and gas limits.
- Caching block headers and receipts to improve retrieval speed.
- Updating the chain's average gas price.

To provide these functionalities, the implementation employs several sub-components, including:

- A consensus component responsible for validating new blocks and incorporating them into the chain.
- A database component that persistently stores blockchain data.
- An event component that notifies other components of chain reorganizations and new block additions.
- A transaction signer component that verifies transaction signatures and identifies the sender address.

- A gas price calculator component that computes the chain's average gas price.

---

id: erc1155 title: ERC-1155 sidebar_label: ERC-1155 description: "Learn about the predefined token standards on a Supernet." keywords: - docs - Polygon - edge - supernets - erc - erc1155 - bridge

ERC-1155 is a token standard that enables the creation of both fungible and non-fungible tokens on a blockchain network. In the context of Supernets, ERC-1155 tokens can be used to represent multiple instances of a single fungible or non-fungible asset on a Supernet.

ChildERC1155

Developers can use the `ChildERC1155` token standard to create ERC-1155 tokens on a Supernet. To create a new ChildERC1155 token, developers can use the `ChildERC1155` contract as a template and deploy it on the Supernet. The contract requires a name and symbol to determine the unique identity of the token.

Once created, `ChildERC1155` tokens can be minted and burned on the Supernet through the corresponding ERC1155Predicate contract. This ensures that the supply of tokens on the rootchain and the Supernet remains in sync. The `ERC1155Predicate` contract also allows for the transfer of tokens between the two.

Predicates

The `RootERC1155Predicate` contract serves as the root token contract, while the `ChildERC1155Predicate` contract serves as the child token contract. When a user wants to deposit an ERC-1155 token on the rootchain into the Supernet, they call the deposit function on the `RootERC1155Predicate` contract with the appropriate parameters, including the token ID, sender, and receiver addresses.

Once the child contract instance is created, the `RootERC1155Predicate` contract maps the root token to the child token using a mapping, allowing users to deposit and withdraw tokens between the root and Supernet. The ChildERC1155Predicate contract includes functions for minting and burning child tokens, which are triggered when the deposit and withdrawal functions are called.

To withdraw a child token back to the rootchain, a user calls the appropriate function on the ChildERC1155Predicate contract with the necessary parameters, including the token ID, the amount to withdraw, and the receiver address. The `ChildERC1155Predicate` contract then burns the child token and emits a withdrawal event. The `RootERC1155Predicate` contract listens for these events and relays them to the rootchain using the `L2StateSender` contract.

Deposits and Withdrawals

Depositing and withdrawing ERC-1155 tokens between the rootchain and Supernet can be done using the `deposit` and `withdraw` functions in the `RootERC1155Predicate` and `ChildERC1155Predicate` contracts, respectively. The `ChildERC1155Predicate` contract includes functions for minting and burning child tokens, which are triggered when the deposit and withdrawal functions are called. To withdraw a child token back to the rootchain, a user calls the appropriate function on the `ChildERC1155Predicate` contract with the necessary parameters, including the token ID, the amount to withdraw, and the receiver address. The `ChildERC1155Predicate` contract then burns the child token and emits a withdrawal event.

id: erc20 title: ERC-20 sidebar_label: ERC-20 description: "Learn about the predefined ERC-20 token standards for Polygon Supernets." keywords: - docs - polygon - supernets - erc - erc20 - token - native

Supernets provide developers with two standards for creating ERC-20 fungible tokens on their networks: `NativeERC20` and `ChildERC20`. These standards are based on the widely-used ERC-20 standard and offer similar functionality, including the ability to transfer tokens between addresses, approve others to spend tokens on your behalf, and check your balance. However, there are significant differences in their deployment and management.

ChildERC20

The `ChildERC20` token standard is used for tokens mapping and enables developers to create tokens on both the Supernet and the associated rootchain. To create a new `ChildERC20` token, developers can use the `ChildERC20` contract as a template and deploy it on their Supernet. The contract requires a name, symbol, and number of decimals to determine the minimum unit of the token.

`ChildERC20` tokens are minted and burned on the Supernet through the corresponding `ERC20Predicate` contract. This ensures that the supply of tokens on the rootchain and the Supernet remains in sync. The `ERC20Predicate` contract also allows for the transfer of tokens between the two networks using the native bridge.

NativeERC20

The `NativeERC20` token standard represents native tokens on the Supernets and offers fast and inexpensive transactions. It is deployed only on the Supernet and relies on the native transfer precompile to make transfers. The `NativeERC20` tokens can be minted and burned by the associated predicate contract.

In addition, developers can use the `NativeERC20Mintable` contract to create and manage `NativeMintable` tokens, which are fungible tokens that represent assets on the Supernet. These tokens can be managed through the native bridge contract and transferred between a Supernet and rootchain networks.

Deposits and Withdrawals

The deposit and withdrawal functionality plays a critical role in bridging ERC-20 tokens between a rootchain and a Supernet.

When a user wants to deposit ERC-20 tokens into a Supernet, they call the `deposit` function. This function maps the root token to a child token and then mints the equivalent amount of child tokens to the user's address on the Supernet. By doing this, the ERC-20 tokens are effectively transferred from the rootchain to the Supernet. The user can then use these tokens on the Supernet or transfer them to other addresses on the network.

On the other hand, when a user wants to withdraw ERC-20 tokens from a Supernet to the rootchain, they call the `withdraw` function. This function burns the equivalent amount of child tokens on the Supernet and then triggers a function on the rootchain that transfers the equivalent amount of root tokens to the user's address on the rootchain. By doing this, the ERC-20 tokens are effectively transferred from the Supernet back to the rootchain.

Both the `deposit` and `withdraw` functions emit events that log the amount and token involved in the transaction. This provides transparency and allows users to track the movement of their tokens between the two networks.

EIP1559Burn

The dynamic gas fee mechanism implements a base fee per gas to improve transaction fee predictability and efficiency. The base fee adjusts dynamically based on gas target deviations and is burned within the protocol.

- **Burn Contract**: The burn contract parameter specifies the contract address where fees will be sent and utilized. To enable the London hard fork and set the burn contract address, use the `--burn-contract` flag in the genesis command with the following format: `<block>:<address>`.
- **Genesis Base Fee**: The genesis base fee parameter sets the initial base fee (in GWEI) for the genesis block. Include this value in the genesis.json file during the genesis command. By default, the genesis base fee is set to 1 GWEI.
- **Genesis Base FeeEM**: The genesis base fee elasticity multiplier determines the base fee for blocks following the genesis block. It is initially set to 2. If the London hard fork is disabled (i.e., when the burn contract is not set), the base fee remains the same as the genesis base fee.
- **Burn Contract Parameter Revision**: The burn contract address can be adjusted during a node reboot, which is achieved by modifying the `burnContract` parameter in the genesis.json file. It's crucial to comprehend, however, that merely identifying the burn contract address does not implicitly activate the EIP-1559 feature. If the EIP-1559 was not initialized during genesis, a series of procedures are required to activate it subsequently:
- The burn contract should be properly deployed onto the Supernet.
- The genesis file must detail the burn contracts map, incorporating both the block number and the burn contract address.
- The base fee and base fee elasticity multiplier are required to be manually defined in the genesis file.
- Finally, the EIP1559 fork must be activated.

---

id: erc721 title: ERC-721 sidebar_label: ERC-721 description: "Learn about the predefined token standards on a Supernet." keywords: - docs - Polygon - edge - supernets - erc - erc721

ERC-721 is a token standard that allows for the creation of unique, non-fungible tokens on a blockchain network. In the context of Supernets, ERC-721 tokens can be used to represent unique assets on the network's Supernet.

ChildERC721

Developers can use the `ChildERC721` token standard to create ERC-721 tokens on the Supernet. To create a new ChildERC721 token, developers can use the ChildERC721 predicate contract as a template and deploy it on the Supernet. The contract requires a name and symbol to determine the unique identity of the token.

Once created, `ChildERC721` tokens can be minted and burned on the Supernet through the corresponding `ERC721Predicate` contract. This ensures that the supply of tokens on the rootchain and the Supernet remains in sync. The `ERC721Predicate` contract also allows for the transfer of tokens between the two networks using the native bridge.

Predicates

The `RootERC721Predicate` contract serves as the rootchain token predicate contract, while the `ChildERC721Predicate` contract serves as the Supernet token predicate contract. When a user wants to deposit an ERC-721 token on the rootchain into the Supernet, they call the `deposit` function on the `RootERC721Predicate` contract with the appropriate parameters, including the token ID, sender, and receiver addresses.

When a user deposits a token for the first time, the token mapping between the root and Supernet is created automatically. This mapping allows users to deposit and withdraw tokens between the root and Supernet.

Deposits and Withdrawals

The `ChildERC721Predicate` contract includes functions for minting and burning child tokens, which are triggered when the deposit and withdrawal functions are called. To withdraw a child token back to the rootchain, a user calls the appropriate function on the `ChildERC721Predicate` contract with the necessary parameters, including the token ID and receiver address. The `ChildERC721Predicate` contract then burns the child token and emits a withdrawal event. The `RootERC721Predicate` contract listens for these events and relays them to the rootchain using the `L2StateSender` contract.

id: checkpoint title: Ensuring Data Integrity through Checkpoint Management sidebar_label: Checkpoints description: "An overview of the checkpointing mechanism of Supernets." keywords: - docs - polygon -

edge - bridge - fxportal - checkpoint

Introduction

Checkpointing is the process of recording and committing a snapshot of the state of a system at a specific point in time. In the context of blockchain, checkpoints are used to increase security by allowing faster verification of the state of the blockchain without needing to process all transactions from the genesis block. Checkpoints can be used to speed up sync times for new nodes and help prevent certain types of attacks, such as 51% attacks.

Checkpoints in Supernets

:::info Key points

- The checkpoints are made **by the validators on the Supernet** and are committed to the rootchain.
- A checkpoint serves as a snapshot of the Supernet state. This snapshot is stored as a Merkle root and **represents the state of the Supernet** at that point in time it was created.
- The checkpoint process is important for **ensuring the security of the network** as it enables the rootchain to detect and prevent any potential fraud or malicious activity on the child.

:::

CheckpointManager

A `CheckpointManager` contract responsible for managing checkpoints in the network.

The checkpoints represent a snapshot of the Supernet state, which is periodically checkpointed to the rootchain by the validators. The checkpoints are used as a reference point for the rootchain to verify the integrity and accuracy of the data on the Supernet.

The contract has several functions to facilitate the management of checkpoints, such as submitting a new checkpoint with metadata, verifying signatures, and getting the event root by block number or epoch. The contract also has a mapping to store the checkpoints and the current validator set, and an array to keep track of the checkpoint block numbers.

The contract uses a Merkle tree to efficiently prove the membership of an event in the Supernet state. The tree is constructed using the hashed exit events sent by the L2StateSender, which the Edge client saves to its local storage after their transactions are executed. The membership proofs can be verified using the Merkle proofs provided by the users.

The contract also implements a BLS signature scheme to verify the signatures submitted by the validators. The validators' signatures are aggregated, and the contract checks whether the required voting power threshold is met to accept the checkpoint.

▶ Details of the checkpoint

---

id: overview title: Introduction to Native Supernet Bridging sidebar_label: Overview description: "Learn about the native bridge of Supernets." keywords: - docs - polygon - bridge - Supernet - rootchain - asset - cross chain

Overview

Supernets provide a built-in bridging mechanism that enables cross-chain communication. The bridging mechanism is a technical infrastructure that facilitates the transfer of arbitrary messages between any EVM-compatible PoS blockchain (rootchain), and a Supernet.

It relies on mapping between the token contracts on the rootchain and the target chain, which is crucial for tracking assets and ensuring the correct amount of tokens are minted and burned during the transfer process.

:::caution Bridge as an Attack Vector The cross-chain bridge can be an attack vector if not properly managed. Ensure you fully understand its functionality, potential vulnerabilities, and security measures before use. Expertise in this area is crucial for maintaining system security. :::

During the transfer process, assets are locked in a contract on the rootchain, and an equivalent amount of tokens are minted on the target chain. When assets are withdrawn from the target chain, the corresponding tokens are burned, and the assets are unlocked on the rootchain. Smart contracts on both the rootchain and the target chain are used to facilitate these processes, ensuring that the asset transfer is secure and transparent.

:::note Enabled by Default

The cross-chain bridge in Supernets is an integral part of the consensus client and is enabled by default. It cannot be disabled, as it is required for the functioning of the network. Being native to the Supernets, the bridge is built into the protocol and does not require any external third-party solutions or plug-ins.

If you choose to use another cross-chain bridging mechanism, you will need to customize it yourself, and there are no guarantees of compatibility with associated contracts and infrastructure.

:::

bridge

The following diagram above provides a visual representation of how messages can be passed between different EVM blockchain layers, allowing for seamless message tranfers and coordination between various components of a Super network.

How does message passing work?

StateSync: real-time synchronization

Message passing between a rootchain and a Supernet is achieved through continuous state synchronization, known as StateSync. This process involves transferring state data between system calls.

Check out the StateSync document to learn more.

Checkpoints: Ensuring liveliness and reference points

When passing messages from a Supernet to a rootchain, the validator set commits checkpoints, which are snapshots of the Supernet state containing only the root of the Exit events, excluding all transactions. Checkpoints serve as reference points for clients, and validators periodically checkpoint all transactions occurring on the Supernet to the rootchain. Checkpoints also ensure liveliness and are submitted to the associated rootchain asset contract.

Check out the Checkpoint document to learn more.

Bridge States: tracking event progress

The bridge can exist in one of three states:

- **Pending**: Events are waiting to be bundled and sent over.
- **Committed**: Event data has been relayed to the associated chain.
- **Executed**: The event has been committed, and the state executed, resulting in a state change.

Assets: supporting a range of standards

The bridge is compatible with various token standards, including ERC-20, ERC-721, and ERC-1155, enabling a wide range of assets to be transferred between chains.
--- id: statesync title: Synchronizing Chain State between Rootchain and Supernet sidebar_label: StateSyncs description: "Learn about the StateSync mechanism of Supernets." keywords: - docs - polygon - supernet - bridge - fxportal - Supernet - rootchain - statesync ---

Introduction

State synchronization (StateSync) is a mechanism used to update the state of a contract on the Supernet based on events occurring on the rootchain. It is a critical component of blockchain technology as it enables secure and efficient communication between the two chains. State synchronization allows for a more efficient and secure way to update the chain state on the Supernet without needing to process all transactions from the genesis block.

StateSync in Supernets

:::info Key points

- `StateSync` enables the efficient and secure transfer of data between a Supernet and rootchain.
- `StateSync` is initiated on the Supernet through the `StateSender` contract and executed on the rootchain through the `StateReceiver` contract.
- The `StateSync` process is used to update the state of a contract on the rootchain based on events occurring on the Supernet. :::

StateSender

The `StateSender` contract is deployed on the rootchain and is triggered by either the associated rootchain predicate or `SupernetManager` rootchain contract. Its main responsibility is to generate sync state events based on the provided data and receiver address. Anyone can call the `syncState` function to emit a sync state event. The data is sent along with the event and represents the state change that needs to be executed on the Supernet.

StateReceiver

The `StateReceiver` contract is deployed on the Supernet and is responsible for executing and relaying the state data sent from the rootchain. It receives the state change data from the rootchain contract bundled up in the form of a commitment, sent with the Merkle Tree root hash. This tree is created by bundling a number of `StateSync` events received by the `StateSender`. Commitments are submitted to the `StateReceiver` by a block proposer, and it is a system (state) transaction. They are used to verify the execution of state data from the rootchain to the Supernet, such as transferring funds from rootchain to Supernet. Commitments are similar to checkpoints but are used in the process of transferring data from rootchain to Supernet, while checkpoints are used in the process of transferring data from Supernet to rootchain.

L2StateSender and ExitHelper

To enable communication from the Supernet to the rootchain, the `L2StateSender` contract resides on a Supernet and is responsible for emitting `L2StateSyncs` (also referred to as exit events). These events are indexed by the rootchain validators and submitted as a checkpoint on the rootchain, allowing for lazy execution. Unlike the `StateSender`, there is no transaction execution on the rootchain for the `L2StateSender`.

On the rootchain, the `ExitHelper` contract is responsible for verifying the execution of the exit events and enabling users to withdraw their funds from the Supernet to the rootchain. It is analogous to the StateReceiver on the Supernet, and both contracts work together to enable two-way communication between the rootchain and the Supernet.

▶ Synchronization and commitments
--- id: ibft-overview title: Istanbul Byzantine Fault Tolerant (IBFT) 2.0 sidebar_label: Istanbul Byzantine Fault Tolerant (IBFT) 2.0 description: "Learn about the consensus engine of Supernets." keywords: - docs - polygon - edge - consensus - polybft - pos ---

Consensus Engine

The PolyBFT consensus mechanism uses the IBFT 2.0 consensus engine to agree on adding new blocks to the blockchain. The validator pool in IBFT 2.0 is responsible for validating candidate blocks proposed by a randomly selected block proposer who is part of the validator pool. The proposer is responsible for constructing a block at the block interval. The proposer mechanism is based on Tendermint, where a proposer is chosen based on a deterministic selection algorithm. The frequency of selection is proportional to the voting power of the validator.

Each block in IBFT 2.0 requires at least one round of voting by the validator to arrive at consensus, which is recorded as a collection of signatures on the block content. In general, a supermajority of validators must confirm that a block is valid for the block to be added to the blockchain. Only when there is no consensus on a given block, multiple rounds of voting are needed. The ideal path would be when the validator pool reaches consensus on a candidate block in the first round of voting, and the block is added to the blockchain without the need for additional rounds of voting. This is the most efficient and optimal outcome, as it allows the network to continue processing transactions and adding new blocks to the chain in a timely manner.

A validator's voting power is proportional to the amount of stake they have locked up on the network. This means that validators with more stake will have more voting power and, therefore, more influence over the decision-making process on the network. This also provides an economic incentive for validators to behave honestly and act in the network's best interest.

▶ Sealing a block

Validator Set

PolyBFT limits network participation to around 100 validators, and a variable amount of stake is used as a fixed stake criterion to limit the system's security and can make the system economically vulnerable. The validator set in the PolyBFT does not update on each block but is fixed during n block periods known as an epoch.

The `n` block period to define one epoch is determined by governance, and until then, validators will remain the same. At the end of the epoch, a special state transaction to validatorSetManagementContract is emitted, notifying the system about the validators' uptime during the epoch. It is up to the smart contract to reward validators by their uptime and update the validator set for the next epoch. There is a function getValidatorSet which returns the current validator set at any time.

:::info Proposer selection algorithm - Section is being updated

The proposer selection algorithm is Tendermint-based.

:::

Staking

Staking is managed by staking contracts on the Supernet. The staking module on Polygon validates staked tokens and is independent of Ethereum's security. In principle, the network is secured by the rootchain and Ethereum. Transaction checkpoints still occur on Ethereum, but Ethereum does not validate staking on Polygon.

At the end of each epoch, a reward calculation occurs to reward validators who actively participated in that epoch.

:::info Staking details and rewards - Section is being updated

:::

Benefits

Consensus benefits

- **Immediate block finality**: Only one block is proposed at a given chain height; thus, the single chain removes forking, uncle blocks, and the risk that a transaction may be "undone" once on the chain later.
- **Reduced time between blocks**: The effort needed to construct and validate blocks is decreased significantly, which increases the chain's throughput.
- **High data integrity and fault tolerance**: IBFT 2.0 uses a pool of validators to ensure the integrity of each proposed block. A supermajority (~66%) of these validators must sign the block before insertion into the chain, making block forgery very difficult. Also, the proposer of the block rotates over time, ensuring a faulty node cannot exert long-term influence over the chain.
- **Operationally flexible**: The validators can be modified quickly, ensuring the group contains only fully trusted nodes.

Network benefits

- **Liveness**: It has been proven that IBFT 2.0 does not guarantee BFT persistence nor liveness when operating on a synchronous network. If a validator receives enough confirmation about a block, it can lock the proposed block (assuming it has not locked any prior). If a change were to occur because of a fault in the network, it could trigger the activation of the round change protocol, where the protocol would expect to commit the locked block at that specific height.
- **Persistence**: If, for instance, there is a faulty network condition present where two different node subsets lock to two different blocks, the system enters into an infinite cycle of state transitions that cannot converge states and finalize the block.

---

id: polybft-overview title: Polygon Byzantine Fault Tolerance (PolyBFT) sidebar_label: Polygon Byzantine Fault Tolerance (PolyBFT) description: "Learn about the new consensus mechanism for the Edge client." keywords: - docs - polygon - edge - consensus - polybft - pos

Overview

PolyBFT is a sophisticated and robust consensus mechanism employed by Polygon Supernets. The consensus mechanism comprises two key components, a **consensus engine** and **consensus protocol**. PolyBFT utilizes the IBFT consensus engine and a Proof-of-Stake architecture to seal blocks, provide specific network capabilities, and govern the network. The core smart contracts work in tandem with the consensus engine to define all the network's Proof-of-Stake rules.

bridge

The consensus engine of PolyBFT is based on the Istanbul Byzantine Fault Tolerance (IBFT 2.0) protocol, which is responsible for sealing blocks on the blockchain. The IBFT 2.0 protocol ensures that network integrity is maintained even in the presence of malicious or dishonest nodes.

To achieve fault tolerance, IBFT allows for `f` faulty nodes in a `3f + 1` network, as long as two-thirds of the nodes are honest. This algorithm is also known as a "super-majority rules" algorithm. Each PolyBFT node maintains a local copy of the blockchain, represented as a list of blocks similar to the blockchain. The height of a block is defined as the number of parent links that separate the block from the genesis block, with the genesis block having a height of 0. Sequential instances of a block finalization protocol are run, with the objective of each instance being to determine which block is to be added at height h of the blockchain.

PolyBFT's consensus protocol is implemented through a set of core smart contracts. These contracts serve multiple purposes, including enabling staking functionality and defining an incentivization scheme for validators on the network, managing the validator set, and facilitating cross-chain communication through a native bridge.

bridge

Bridge

The bridge transfers assets and data between a Supernet and an external EVM-compatible blockchain (rootchain), making it a critical component of the network's interoperability. Two predicate contracts, one on the childchain and one on the rootchain, implement the core bridge functionality and use the associated core contracts to deposit, withdraw, and verify cross-chain bridge transactions. The diagram below illustrates how the core contracts fit into the overall smart contract system design.

id: polybft-protocol title: Consensus Protocol sidebar_label: Consensus protocol description: "Explanation about consensus protocol within PolyBFT." keywords: - docs - polygon - edge - consensus - polybft - pos

State Transitions

In PolyBFT, the consensus protocol follows a set of state transitions that enable the network to reach consensus on the state of the blockchain. The process begins when a validator proposes adding a new block to the blockchain. This block contains a list of transactions that the validator would like to include in the next update to the blockchain's state.

Other validators in the active set will vote on whether to accept the proposed block. Each validator has a voting weight that influences voting. A super-majority of validators must agree to accept the block to reach consensus. Once a sufficient number of validators have agreed to accept the proposed block, it is added to the blockchain, and the state of the blockchain is updated to reflect the changes introduced by the transactions in the block. The protocol refers to the current block's position (the block height) as its *sequence*.

The process to finalize a block in PolyBFT is known as *sealing*. When a validator proposes a new block, other validators on the network will vote on whether to accept the block. This process is may be repeated several times; each repetition is known as a *round*. During each round, certain validators must agree to seal the proposed block for it to be added to the blockchain. Another validator will attempt to seal the sequence in the new round. If the required number of votes is not reached during a particular round, the voting process will continue into the next round, and thus, the protocol "increases the round".

Once the proposed block is accepted, it will be added to the blockchain, and the state of the blockchain will be updated to reflect the changes introduced by the transactions in the block. This process continues with the next proposer proposing a new block, and the process repeats.

Validator Set

PolyBFT limits network participation to around 100 validators, and a variable amount of stake is used as a fixed stake criterion to limit the system's security and can make the system economically vulnerable. The validator set in the PolyBFT does not update on each block but is fixed during n block periods known as an epoch.

The `n` block period to define one epoch is determined by governance, and until then, validators will remain the same. At the end of the epoch, a special state transaction to validatorSetManagementContract is emitted, notifying the system about the validators' uptime during the epoch. It is up to the smart contract to reward validators by their uptime and update the validator set for the next epoch. There is a function getValidatorSet which returns the current validator set at any time.

Staking

Staking is governed by staking contracts directly on Polygon. To be clear, the staking module validates on Polygon and does not rely on Ethereum's security, but in principle, two chains are securing the network, PoS client and Ethereum. Transaction checkpoints still occur on Ethereum, but Ethereum does not validate staking on Polygon.

At the end of each epoch, a reward calculation occurs to reward the active validators.

Native Bridge Integration

PolyBFT supports an in-built bridging mechanism that enables arbitrary message passing between a Supernet and another Proof-of-Stake blockchain (rootchain) with the help of the consensus algorithm. Transfers can occur without mapping, allowing users to move assets between the two chains seamlessly.

Slashing

Like in other Proof-of-Stake systems, validators are subject to slashing for malicious activity or poor performance. The slashing mechanics are still being determined, but PolyBFT will undoubtedly include a mechanism to penalize bad actors. Slashing a validator typically involves a penalty, such as losing some or all of their stake on the network. Examples of malicious activities are double-signing and equivocation.

State transitions

The consensus protocol follows a set of state transitions. While things are still being finalized, the process will typically follow the steps below.

1. A validator proposes adding a new block to the blockchain. This block contains a list of transactions that the validator would like to include in the next update to the blockchain's state.

A certain number of validators must agree to accept the block to reach consensus. 2. Other validators in the active set will vote on whether to accept the proposed block. The voting weight of each validator influences voting. The protocol refers to block height as a *sequence*.

```
All nodes in the network exchange information for a given sequence. The process to finalize a block in PolyBFT is known as
*sealing*. The sealing of blocks is instant
```

```
and final.

When a validator proposes a new block, other validators on the network will vote on whether to
accept the block. This process is typically repeated several times; each repetition is known as a
*round*. During each round, certain validators must agree to seal the proposed block
for it to be added to the blockchain. If the required number of votes is not reached during a
particular round, the voting process will continue into the next round, and thus, the protocol
"increases the round". Another validator will attempt to seal the sequence in the new round.
> The best case for a proposed block is that it is sealed at round 0. Blocks that are repeatedly
> sealed at a high-order round usually indicates a network problem.
```

3. If the proposed block is accepted, it will be added to the blockchain, and the state of the blockchain will be updated to reflect the changes introduced by the transactions in the block.

   > *If a malicious actor attempted to fork the network, they would need to obtain control of 2/3 of the network, which PolyBFT prevents.*

4. Once the state of the blockchain has been updated, the next proposer will propose a new block, and the process repeats.

IBFT limits network participation to around 100 validators. A variable amount of stake is used as a fixed stake criterion to limit the system's security and can make the system economically vulnerable. The validator set in the PolyBFT does not update on each block but is fixed during `n` block periods known as an `epoch`.

> *The `n` block period to define one epoch is to be determined by governance. Until then, validators will remain the same. At the end of the epoch, a special `state transaction` to `validatorSetManagementContract` is emitted, notifying the system about validators' uptime during the `epoch`. It is up to the smart contract to reward validators by their uptime and* ***update the validator set*** *for the next `epoch`. There is a function `getValidatorSet`, which returns the current validator set at any time.*

Staking is governed by staking contracts directly on Polygon. To be clear, the staking module validates on Polygon does not rely on Ethereum's security, but in principle, two chains are securing the network, PoS client and Ethereum. Transaction checkpoints still occur on Ethereum, but Ethereum does not validate staking on Polygon.

:::note

Note that in Tendermint, an epoch is set to 1. However, PolyBFT includes the logic to set a custom epoch time, with the intent of each epoch being one day in blocks, or around 14000 blocks.

:::

A reward calculation occurs at the end of the epoch to reward the active validators in that epoch.

:::caution Slashing

Like in other Proof-of-Stake systems, validators are subject to slashing for malicious activity or poor performance. Slashing a validator typically involves a penalty, such as losing some or all of their stake on the network. The slashing mechanics are still being determined, but PolyBFT will undoubtedly include a mechanism to penalize bad actors.

Examples of malicious activities are double-signing and equivocation:

- Double-signing refers to the act of signing two conflicting transactions. When a validator double-signs, it creates a situation where the network is unable to reach consensus on the state of the blockchain, which can lead to problems such as an attempt to fork or network instability.
- Equivocation refers to the act of a validator attempting to create two conflicting versions of the blockchain, which can also lead to problems such as a fork or network instability.

:::

Block timestamp drift validation

In the PolyBFT consensus mechanism, an important security feature is the validation of block timestamps. This feature prevents malicious actors from creating blocks with timestamps far in the future, which could disrupt the blockchain's operation.

The block time drift validation works by checking if the block's timestamp is within a predefined time slot. If the timestamp is outside this window, the block is considered a future block and is rejected. This time slot is defined by a parameter called `blockTimeDrift`, which is nominated in seconds, and can be set during the genesis of the blockchain. The default value for `blockTimeDrift` is 1 second.

This feature ensures that all blocks in the blockchain have valid timestamps, preventing potential attacks and maintaining the integrity of the blockchain.

Native bridge integration

With the help of PolyBFT, Supernets support an in-built bridging mechanism (a two-way bridge), which enables arbitrary message passing between a Supernet and another Proof-of-Stake blockchain (`rootchain`). Transfers can occur without mapping.

bridge

Learn more here.

id: supernets-grpc title: gRPC description: "The gRPC layer fof Supernets." keywords: - docs - polygon - edge - stake - unstake - validator - epoch

What is gRPC used for?

gRPC is a popular RPC (remote procedure call) framework that enables efficient communication between distributed systems. It is commonly used in microservices architectures and is especially useful for building high-performance, scalable applications that require low-latency communication between services. With its support for multiple programming languages and its built-in support for data serialization, gRPC provides a powerful and flexible tool for building distributed systems.

gRPC in Supernets

libp2p provides the underlying networking layer for establishing connections between peers. By creating a gRPC server instance on each peer in the network and allowing gRPC client connections to be established, a network API can be created that enables clients to interact with the network.

When a client makes a remote procedure call to the server, the gRPC protocol abstracts the details of network communication, serialization, and deserialization. The gRPC client sends a request message to the gRPC server, which deserializes the request message, executes the appropriate method, and serializes the response message. The gRPC server then sends the response message back to the client, which deserializes the response message and returns the result to the caller.

▶ Breakdown of the gRPC API
--- id: supernets-json-rpc title: JSON-RPC sidebar_label: JSON-RPC description: "The JSON-RPC layer of Supernets." keywords: - docs - polygon - edge - architecture - modular - layer - libp2p - extensible ---

What is JSON-RPC used for?

JSON-RPC is a remote procedure call (RPC) protocol encoded in JSON. It enables the communication between distributed systems, often found in microservices architectures. JSON-RPC is a lightweight, language-agnostic protocol that allows developers to build scalable applications with low-latency communication between services. With its simplicity and support for multiple programming languages, JSON-RPC offers an accessible and flexible tool for building distributed systems.

JSON-RPC in Supernets

JSON-RPC is implemented in Supernets as an API consensus.

When a client makes a remote procedure call to the server, the JSON-RPC protocol abstracts the details of network communication, serialization, and deserialization. The JSON-RPC client sends a request message to the JSON-RPC server, which deserializes the request message, executes the appropriate method, and serializes the response message. The JSON-RPC server then sends the response message back to the client, which deserializes the response message and returns the result to the caller.

▶ Breakdown of the JSON-RPC API
--- id: supernets-libp2p title: libp2p sidebar_label: Networking description: "The networking layer of Supernets." keywords: - docs - polygon - edge - stake - unstake - validator - epoch ---

Overview

PolyBFT uses a decentralized networking layer based on the libp2p protocol. The protocol provides peer-to-peer networking primitives such as peer discovery, connection management, and secure messaging. The network relies on a secure Identity Service to manage peer connectivity and handshaking, ensuring only valid peers can join the network.

Identity

The Identity Service validates incoming connections and manages peer handshaking. It maintains a list of pending peer connections and uses a networkingServer interface to communicate with the underlying networking layer.

Peer discovery

PolyBFT uses libp2p's distributed hash table (DHT) based on the Kademlia algorithm for peer discovery. The DHT stores information about other peers in the network, such as their addresses and availability. When a new node joins the network, it uses the DHT to find other peers that are currently online. The process of using the DHT to discover peers and then sending out connection requests is repeated periodically to maintain a sufficient number of connections in the network.

Peer routing

Bootnodes act as rendezvous servers that help new nodes discover and connect to the network. The polygon-edge command allows you to specify one or more bootnodes while creating the genesis file. Bootnodes are defined using libp2p multiaddrs, which contain information about the protocol, network address, and node port number.

```
--bootnode /ip4/127.0.0.1/tcp/30301/p2p/16Uiu2HAmJxxH1tScDX2rLGSU9exnuvZKNM9SoK3v315azp68DLPW \
--bootnode /ip4/127.0.0.1/tcp/30302/p2p/16Uiu2HAmS9Nq4QAaEiogE4ieJFUYsoH28magT7wSvJPpfUGBj3Hq \
```

Gossipsub

Gossipsub is a decentralized, peer-to-peer messaging protocol used in Polygon Supernets to broadcast messages efficiently across the network. It's used in various network components, including the TxPool, where it's used to broadcast new transactions and relay transaction data between nodes. Gossipsub allows for efficient and reliable message propagation while minimizing the bandwidth requirements of the network.

To learn more about libp2p networking, check out the [official libp2p documentation](#).

id: supernets-mempool title: Memory pool sidebar_label: Memory Pool description: "The memory pool of Supernets." keywords: - docs - polygon - edge - architecture - modular - layer - libp2p - extensible

What is a Mempool?

A mempool, short for "memory pool," is a temporary storage area in a blockchain node for pending transactions before they are added to a block and confirmed on the network. When users submit transactions, they are initially held in the mempool, where miners or validators can select them to include in the next block. Transactions in the mempool are generally ordered based on factors such as transaction fees and age, which helps prioritize transactions for faster confirmation.

Mempool in Supernets

In Supernets, the mempool is crucial in managing and prioritizing pending transactions. When a transaction is submitted to the network, it is first stored in the mempool, which remains until it is confirmed in a block. The mempool helps ensure that transactions are executed fairly and efficiently, as validators can choose which transactions to include in a block based on factors such as fees and age.

The mempool in Supernets is designed to optimize transaction processing and improve network performance. Some of the key features of the mempool in Supernets include:

- **Transaction prioritization**: The mempool prioritizes transactions based on fees and age. Transactions with higher fees are generally processed faster, providing greater incentives for validators to include them in a block.
- **Mempool management**: Mechanisms for handling congestion, transaction eviction, and memory usage are implemented to help maintain a healthy mempool, ensuring that the network remains stable and responsive.
- **Transaction validation**: Before a transaction is added to the mempool, validators validate it to ensure that it is valid and adheres to the network's consensus rules.
- **Mempool synchronization**: Supernet nodes share information about their mempools to maintain a consistent view of the pending transactions in the network.

---

id: overview title: Architecture Overview sidebar_label: Overview description: "Introduction to the architecture of Supernets." keywords: - docs - polygon - edge - architecture - modular - layer - libp2p - extensible

The diagram below illustrates the major coomponents of Supernets.

Supernets architecture overview

Components

The following table breaks down each of these components.

Component	Description	Link
libp2p	Provides the networking layer for Supernets and is designed for peer-to-peer network architectures.	libp2p overview
Bridge	An in-built bridging mechanism enabled by PolyBFT that allows message passing between a Supernet and another Proof-of-Stake blockchain without mapping.	bridge overview
Mempool	Enables multiple validators to aggregate their signatures to create a single, aggregated signature representing all validators in the pool.	mempool overview
Consensus	PolyBFT serves as the consensus mechanism of Polygon Supernets and consists of a consensus engine, IBFT 2.0, and a consensus protocol that includes the bridge, staking, and other utilities.	PolyBFT overview
Blockchain	Coordinates everything in the system, curates state transitions, and is responsible for state changes when a new block is added to the chain.	blockchain overview
Runtime (EVM)	Uses the EVM as the runtime environment for executing smart contracts.	runtime overview
TxPool	Represents the transaction pool, closely linked with other modules in the system.	txpool overview
JSON-RPC	Facilitates interaction between dApp developers and the blockchain, allowing developers to issue JSON-RPC requests to a Supernet node and receive responses.	JSON-RPC overview
gRPC	Essential for operator interactions, allowing node operators to interact with the client easily and providing a seamless user experience.	gRPC overview

---

id: supernets-runtime-allowlist title: Access Control Lists sidebar_label: Access control lists description: "Learn about the authorization capabilities available in Supernets." keywords: - docs - polygon - allowlist - blocklist - acl - bridge - contract - transaction

Overview

Supernets offer access control lists (ACLs) to manage and control access to specific network resources, contracts, or functionalities. Network operators can limit access to certain addresses using ACLs, ensuring a controlled and secure environment for their applications.

Roles

There are currently two primary roles within the ACL system:

- **Admin**: An address with full control over the list, capable of adding, modifying, or removing entries.
- **Enabled**: An address that has been granted access to the specific resource or functionality covered by the list.

Initializing and Managing Admins in ACLs

To effectively manage ACLs, at least one address with the Admin role should be assigned when initializing each list in your blockchain application. Admins have the authority to modify the roles of other addresses, granting or revoking access to resources or functionalities as needed.

When setting up ACLs, you should:

- Select trusted addresses to serve as admins.
- Assign the `Admin` role to these addresses using the appropriate flag for each list type.
- Admin addresses will manage the respective ACLs, including adding or removing new addresses. Ensure that the admin addresses are secure and that only authorized users have access to them to maintain the integrity and security of your application.

Designating Multiple Admins

You can designate multiple admin addresses for redundancy and better access control management. Having more than one admin can help:

- Prevent single points of failure.
- Distribute the responsibility of managing the ACLs among multiple trusted entities.

However, assigning multiple admins also increases the potential attack surface. It is essential to strike the right balance between security and flexibility when managing admin addresses or use your best judgment based on the needs and purpose of the network.

ACL Types

Supernets support several types of ACLs:

- **Contract Deployer Allow/Block Lists**: Controls which can deploy contracts on the network.
- **Transactions Allow/Block Lists**: Controls which addresses can send transactions on the network.
- **Bridge Allow/Block Lists**: Controls that can interact with the bridge functionality.

Each list type serves a different purpose and can be configured separately to provide fine-grained control over various aspects of the network. Network operators can use a combination of these lists to maintain a secure and controlled environment for their applications.

Bridge ACLs

Bridge ACLs are an additional layer of security for cross-chain transactions, allowing only specific authorized contracts to interact with a particular bridge. By using an alternative ACL-enabled contract, network operators can have greater control over cross-chain interactions, ensuring that only trusted parties can perform certain actions.

However, it's essential to note that using an alternative ACL-enabled contract comes with a trade-off in terms of gas consumption. Since these contracts introduce additional checks and restrictions on transactions, they require more computational resources, leading to higher gas fees. Network operators must carefully weigh the benefits of increased security against the potential increased costs for users.

Current Limitations

- **Limited role definitions**: The current implementation only supports two primary roles (Admin and Enabled). You must modify the implementation if your application requires more granular access control or additional roles.
- **Granularity**: The current implementation applies the ACLs at the network level. More granular control may require custom implementation, such as using ACLs for specific contracts or functionalities.
- **Static call limitation**: Write operations are not allowed in static calls, which might be a constraint in specific scenarios where you would want to perform a write operation within a static call.

---

id: supernets-runtime title: Ethereum Virtual Machine (EVM) sidebar_label: Overview description: "Learn about the runtime environment of Supernets." keywords: - docs - polygon - edge - architecture - modular - layer - libp2p - extensible

The EVM is the core interface for executing and interacting with smart contracts on a Supernet. It operates consistently with the Ethereum EVM and supports the same set of OPCODEs.

▶ Overview of the EVM

--- id: supernets-txpool title: Transaction Pool sidebar_label: Transaction Pool description: "The transaction pool of Supernets." keywords: - docs - polygon - edge - architecture - modular - layer - libp2p - extensible ---

What is the TxPool used for?

The TxPool module manages the incoming transactions for processing. It maintains a list of unprocessed transactions and ensures they conform to specific constraints before entering the pool.

TxPool in Supernets

The TxPool has several methods to handle incoming transactions.

▶ Details on how the TxPool works

--- id: supernets-txrelayer title: Transaction Relayer sidebar_label: Transaction Relayer description: "The transaction relayer of Supernets." keywords: - docs - polygon - edge - architecture - modular - layer - libp2p - extensible ---

What is a TxRelayer used for?

A TxRelayer is a component that facilitates the creation and sending of transactions. It provides a simple interface for executing a message call without creating a dedicated transaction and sending signed transactions to a blockchain.

TxRelayer in Supernets

In Polygon Supernets, the TxRelayer module contains a client communicating with an Ethereum node and a wallet for signing transactions.

▶ Details on using the TxRelayer

--- id: supernets-faq title: Frequently Asked Questions sidebar_label: Frequently Asked Questions description: "The most common questions and answers about Supernets" keywords: - docs - Polygon - edge - supernets - network - modular - faqs - questions ---

import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem';

This document contains answers to frequently asked questions (FAQs) about Supernets.

If you're new to Supernets and need general information, please navigate to the **Common FAQs tab**. If you're ready to deploy a Supernet or have technical deployment questions, please navigate the rest of the **FAQ tabs**. **If you have any further questions about configuring Supernets, please consult the documentation or reach out to the Supernets team for support.**

<Tabs defaultValue="common" values={[{ label: 'Common', value: 'common', }, { label: 'Migration', value: 'migrate', }, { label: 'Node & Validators', value: 'validator', }, { label: 'Assets & Bridge', value: 'assets', }, { label: 'Rootchain', value: 'rootchain', },] }>

What is the relation between Supernets and Edge?

Supernets are built on top of an updated version of the legacy Edge client. This iteration of the client is designed specifically to enable next-generation EVM-compatible, application-specific sovereign blockchains.

:::info Edge + Supernets = Geth + Ethereum

Before diving into the documentation, please keep the following points in mind:

- **The Polygon Labs team will prioritize the latest version of the Edge client and as a result, will no longer provide support for older versions (v0.6.3 and earlier).**

  The Edge repository will remain accessible, and users may fork it and use it as they wish, subject to applicable open-source license terms. It is **highly recommended to upgrade to the latest version**, which includes the most up-to-date features and fixes. However, users who prefer to stay on older versions may continue to do so.

- **Polygon Edge serves as a consensus client implementation for Polygon Supernets, much like how Geth serves as a client implementation for Ethereum.** Both Geth and Edge serve as intermediaries between nodes and their respective blockchains, allowing users to interact with the network and take advantage of its benefits. To draw an analogy, Geth and Edge are to Ethereum and Supernets what web browsers are to the internet, enabling users to access and interact with the network.

:::

Why should I use Supernets instead of Edge?

Supernets are a product suite; a more advanced and feature-rich version of the Edge client that are specifically designed for building and deploying sovereign blockchains. It includes additional tools, features, and optimizations that make it easier to develop and deploy blockchain applications, while addressing the limitations of the legacy Edge client. Additionally, because the legacy Edge client will no longer be supported, developers are encouraged to transition to Supernets for ongoing support and development.

In addition, please check out the why Supernets document for more context.

What are some use cases for Supernets?

Supernets are highly flexible and customizable blockchain platforms that can be used to create tailored solutions for a wide range of use cases. Here are some examples of potential use cases for Supernets:

- Gaming applications and platforms
- Decentralized finance (DeFi) applications
- Supply chain management solutions
- Identity verification and management systems
- Social networks and content platforms
- Tokenization of assets such as real estate, art, and intellectual property
- Decentralized autonomous organizations (DAOs) and governance systems
- IoT and machine-to-machine communication
- E-commerce platforms and payment systems
- Insurance and risk management solutions

Is Edge the only consensus client for Supernets?

Yes, Edge is the sole consensus client for Supernets and is integrated into the product suite.

What tools are available for Supernets?

Developers will have access to a variety of tools for building and launching blockchains on the Supernets network. Polygon Labs has partnered with multiple infrastructure providers, tool providers, and implementation partners to offer an all-inclusive development experience.

Supernets will eventually provide native tooling supplemented by third-party integrations and support. As the ecosystem grows, more resources are expected to become available, making it even easier for developers to build and deploy blockchain solutions on the Supernets network.

What infrastructure providers support Supernets?

Several infrastructure providers are currently onboarded to support Supernets and more are expected to join in the future. These providers offer a range of services including node hosting, developer tools, and technical support to help developers build and deploy applications on the Supernets network.

In addition to providing infrastructure and technical support, many of these providers also offer their own resources and documentation to aid in the utilization of Supernets. As the Supernets ecosystem continues to grow, we expect to see even more infrastructure providers joining the platform, further expanding the resources and support available to developers.

Can you use Supernets with native Ethereum tooling?

Yes, Supernets are EVM compatible and support Ethereum libraries such as Web3.js and Ether.js, making it easy for developers familiar with the Ethereum ecosystem to build and deploy applications on the Supernet. Additionally, popular Ethereum development environments such as Remix and Truffle can also be used with Supernets.

Do Supernets address the need for large blockspace requirements?

Yes, Supernets are designed to meet the need for large blockspace requirements by providing a highly scalable, customizable blockchain solution. Developers can create application-specific blockchains that cater to their unique needs by extending the blockspace of a rootchain. The PolyBFT consensus mechanism used by Supernets is specifically designed to provide high throughput, low latency, and instant network finality, making it ideal for a wide range of applications already on Polygon PoS mainnet.

Can the native bridge be deployed on any chain?

Yes, the bridge can connect to any EVM-compatible rootchain (e.g. Polygon PoS). By leveraging the capabilities of the bridge and the modular design of Supernets, developers can create tailored solutions that are customized for their specific needs, including how they utilize assets.

What assets are supported on the bridge?

The bridge currently supports ERC-20, ERC-721, and ERC-1155 tokens on Supernets for transfer. However, support for other asset types will soon be available. It is important to keep in mind that not all tokens on Supernets have the same functionality and capabilities. Users should review the bridge document to understand how to use tokens on Supernets and with the bridge.

As the Supernets ecosystem evolves, the bridge will support additional assets, enhancing interoperability between various blockchain networks and applications.

When will the Supernets mainnet be live?

There is no mainnet for Supernets - Supernets are a product suite. The production release of Supernets is currently **targeted for Q2 2023, subject to the completion of an ongoing audit**.

The development team is committed to ensuring that the platform is thoroughly tested and secure before release. In the meantime, developers and users can experiment with the latest test release, v1.0.0-rc1, which serves as a valuable testing ground for applications on the platform.

When are the audits taking place?

The audits are currently underway and scheduled to conclude in Q2 2023.

What will be the result of the audits?

The audits will assess the security and functionality of the v1.0 release. If any issues are found, they will be addressed before the official release.

How will the results of the audits be communicated if any issues are found?

If any issues are found during the audits, they will be communicated through official channels. **Please stay tuned for more information regarding the audits.**

Can I migrate from an older version of Edge to Supernets?

Yes, it's possible to migrate from a blockchain running on an older version of Edge with IBFT 1.0 consensus to a new blockchain running on Supernets with PolyBFT consensus using IBFT 2.0. However, please note that the migration process will require a hard fork, as IBFT 2.0 is not fully backward compatible with IBFT 1.0.

To upgrade to a new blockchain with PolyBFT consensus using IBFT 2.0, you'll need to follow these steps:

- Create a new genesis block for the new blockchain with PolyBFT consensus using IBFT 2.0.
- Migrate the data and state from the old blockchain to the new blockchain. **Currently, the product suite only support state migration.**

For more information on the migration process, please refer to the migration guide available here.

Does the current implementation support configuration of an RPC node and full node?

Presently, the system does not support such configurations. All nodes function as archive nodes, maintaining the entire blockchain history. The deployment workflow distinguishes only between validators and non-validators.

How can I monitor my nodes?

You can use common tools like Prometheus and Grafana to monitor your nodes. These tools can help you track important metrics like CPU usage, memory consumption, and network activity. Additionally, you can use block explorers like Polygonscan to monitor your node's health and activity.

We will also provide monitoring node guides, as well as tutorials on how to *upgrade* or *migrate* and test shortly.

What is the block interval in Edge?

Edge is configured to generate blocks every second, but the interval between blocks can vary due to the dynamic nature of block production. The `edge_consensus_block_interval` is computed based on the difference between block headers, which have a timestamp field with only 1-second precision. The interval can never be 0 based on the PolyBFT consensus rules.

Why can't two blocks have the same timestamp?

The timestamp field in the block headers only has 1-second precision. Therefore, it's not possible for two blocks to have the same timestamp. Based on the PolyBFT consensus rules, two blocks won't be produced in the same second.

What causes the variability in block production times?

Because the system clock and block production are not phase-locked, there can be slight variations in the time it takes to produce each block. These variations can be caused by a number of factors, such as network latency, differences in computational power between validators, and the availability of resources like memory and disk space. Additionally, PolyBFT ensures that blocks are not produced too frequently or too infrequently, which can also contribute to variations in block production times. Still, the system is designed to ensure that blocks are produced reliably and at a predictable rate.

Is it possible to phase lock the system clock and block production in Edge?

No, it's not possible to phase lock the system because we wouldn't produce a block in less than a second. This dynamic is fundamental to the way that times are being stored in the block headers and the rate at which blocks are being produced. It doesn't speak to the variability of block production times.

What is the minimum and maximum validator size?

The minimum number of validators required is **three** in order to fulfill the requirements of IBFT 2.0 and achieve consensus, as at least two-thirds of the network are required to reach consensus.

However, to achieve fault tolerance in PolyBFT, it is recommended to run at least four validators.

> *PolyBFT does not support single validator architecture at this time.*

The maximum validator size is estimated to be 100, which is currently the upper limit for network stability and performance. However, this may change in the future as the network evolves and undergoes further testing and optimization.

How do validator rewards work?

The reward mechanism in the network incentivizes validators with block rewards for producing valid blocks each epoch. As of now, there is no separate predicate in place to withdraw rewards over the bridge. The details regarding the reward distribution and schedule are still being determined by the development team.

Once the implementation is ready, the specifics will be communicated.

How should I handle validator secrets to ensure network security?

To maintain the security of the network, it is recommended that you keep the validator secrets secure and only retrieve them when necessary. Avoid storing them in an unsecured location or sharing them with unauthorized parties.

Should I consider running a node in "relayer" mode?

Running one Supernet node in "relayer" mode can enable automatic execution of deposit events. This can help streamline the deposit process and improve the overall efficiency of the network.

Can a Supernet use a custom token standard instead of the default token contracts?

Although it is recommended to use the standard ERC-20, ERC-721, and ERC-1155 contracts, it is not a requirement for a Supernet to use them. However, any custom standard employed by a Supernet must adhere to the chain configuration to ensure compatibility with the bridge. Additionally, generating a native ERC-based asset using the default core contracts involves deploying, mapping, and bridging. If a custom standard is used, it will also need to follow the same procedure.

What is the recommended way to create and manage assets on a Supernet?

To create and manage native assets on a Supernet or enable asset bridging between a Supernet and rootchain, it is recommended to use the core contracts provided by the network.

Is it necessary to use the native bridge integration for Supernets?

Yes, it's essential. The native bridge integration is fundamental for Supernets as it hosts the staking logic on the rootchain, which serves as the authoritative source of truth for the Supernet.

While customization is possible, it should be undertaken with a comprehensive understanding of the system. If preferred, the product suite can be used solely for the childchain instance.

Can the gas token be different from the staking token in Supernets?

Yes, Supernets allows for the decoupling of the native gas token and the staking token. You can set any ERC-20 token as your gas token, and use MATIC for staking.

Decoupling the gas token and the staking token provides greater flexibility and enables more use cases for the network. However, it's important to note that the specifics of how this is configured may depend on the specific implementation of the network you're using.

How can I create the initial supply of tokens when launching a new Supernet instance?

To create the initial supply of tokens when launching a new Supernet instance, premining or minting can be used. This can help ensure that the network has the necessary tokens to facilitate transactions and operations.

Why does the Exit process revert when onL2StateReceive fails?

The system prevents potential double spending and other attacks by marking any exit event that fails to be processed by the rootchain predicate as "failed." Without this safeguard, the same event could be re-executed multiple times.

Can checkpoint frequency be influenced or triggered earlier, such as when bridge requests are pending?

Checkpoints are currently dispatched at the end of each epoch or every 900 blocks, a hardcoded value. A future update aims to adjust parameters through a hard fork mechanism. Notably, any participant can submit a checkpoint, as data is produced for each block.

How are native gas fungible tokens allocated during the genesis of the childchain?

At the Supernet's inception, designated accounts receive specific amounts of native gas fungible tokens. If the tokens are non-mintable, premining at genesis is prohibited, except for the 0x0 address. These accounts must bridge assets from the rootchain. Conversely, mintable tokens allow for arbitrary premining on the Supernets during genesis.

Is it necessary to mint new tokens using the associated ERC20 contract each time native gas fungible tokens are spent?

Yes, minting occurs either on the childchain or rootchain, depending on the type of native token (mintable vs non-mintable). For non-mintable tokens, minting should occur on the rootchain, then the tokens should be deposited or bridged to the childchain, creating a new supply.

Is there a mechanism to recover spent gas on the childchain back into an account?

Currently, there's no identified mechanism to recuperate gas spent on the childchain back into an account.

What is the recommended RPC endpoint for the rootchain?

You may use any available JSON-RPC endpoint for an associated rootchain.

However, selecting a reliable and publicly available RPC endpoint for the rootchain is crucial to ensure proper handling of checkpoint information. To guarantee validators can facilitate specific requests, it is suggested to connect directly to a node endpoint instead of a load balancer.

Can existing rootchain contracts be used instead of needing to deploy new ones?

In theory, existing rootchain contracts can be used, but it is not practical with the current contract configuration.

Manually updating the Supernet configuration to point to existing rootchain contract addresses may disrupt or challenge checkpoints and chain state, leading to problems if there is a fork on the Supernet. Furthermore, updating the checkpoint manager's address could trigger an exit with the wrong checkpoint manager, leading to incorrect state transitions and potential loss of funds. Deploying new contracts on the rootchain ensures all necessary dependencies are available and properly configured, avoiding compatibility issues or other unforeseen problems.

Is it necessary to deploy new instances of rootchain contracts for each Supernet instance when running multiple Supernets?

Yes, new instances of rootchain contracts must be deployed for each Supernet instance. Each Supernet instance is independent and requires its own set of contracts to function properly. Deploying and managing multiple instances of contracts can be complex and requires careful consideration of factors such as cost, security, and scalability. It is essential to plan and test thoroughly before implementing a solution with multiple Supernets.

Is the identity of a Supernet instance linked to the rootchain contract address on the rootchain?

Yes, the identity of a Supernet instance is linked to the rootchain contract address on the rootchain. Each Supernet instance is associated with a specific set of rootchain contracts, and its identity is determined by the address of those contracts on the rootchain.

Can an allowlist be added after genesis if it was not configured during chain configuration?

No, it's not possible to add an allowlist after the genesis configuration. This is because the allowlist serves to specify the initial "admins" of the network who are responsible for the initial authority and adding new addresses thereafter. If the allowlist were to be changed after genesis, it would alter the initial authority of the network, potentially introducing security risks and undermining the integrity of the system. As such, it's important to carefully consider and configure the allowlist during the genesis setup to ensure the network's security and functionality.

--- id: what-are-supernets title: What are Polygon Supernets? sidebar_label: What are Supernets description: "An introduction to Polygon Supernets." keywords: -

docs - Polygon - edge - supernets - childchain - network - modular ---

Introduction

Supernets are application-specific chains that operate on the Polygons Edge consensus client with PolyBFT consensus. They leverage a native bridge to connect with an associated rootchain, enabling them to inherit its security and capabilities. Additionally, Supernets extend the block space available on the rootchain, providing scalability and interoperability for decentralized applications. With on-chain governance mechanisms, Supernets empower communities to make decisions and upgrade the network in a transparent and compliant manner.

The following table offers a comprehensive overview on what Supernets are through different perspectives.

Supernets are	Description
Application-specific blockchain networks designed for specific use cases.	Appchains are customizable blockchain networks that developers can tailor to meet specific enterprise or application use cases. With appchains, developers can create high-performance blockchain networks that are fast and low-cost.
The next phase of Polygon PoS and Ethereum scaling.	Supernets empower projects and enterprises to create highly scalable blockchain networks that meet their specific block space requirements while inheriting the security and integrity of the Polygon PoS and Ethereum mainnet.
Modular and complex-minimized blockchain development.	Supernets offer a modular framework that simplifies blockchain development, providing developers with the tools necessary to create customizable blockchain networks that are scalable, secure, and interoperable. Developers can use the Supernets stack to create high-performance blockchain networks with advanced capabilities without worrying about complex integrations or intermediaries.
Customizable blockchain networks for reliable business logic.	A customizable virtual machine provides full Ethereum Virtual Machine (EVM) support out of the box, enabling developers to tailor the virtual machine to their specific needs and requirements. This feature includes customizing gas limits, adding new opcodes, and integrating with other technologies.
The most supported blockchain infrastructure in the web3 space.	The Supernets ecosystem includes the most extensive suite of premium service providers, who offer various node and deployment infrastructure, indexers, explorers, oracles, and many other world-class tools necessary for building and deploying Supernets and their associated applications.
Adaptive and compliant blockchain networks that thoroughly reflect their maintainers.	Supernets feature an on-chain governance mechanism that enables community-driven decision making and management of the blockchain. This mechanism supports hybrid governance models and allows for the design of either permissionless networks or permissioned networks with varying degrees of sovereignty based on user and maintainer needs, including the ability to allowlist validators and network admins.
Interoperable and multichain driven	A native bridging solution enables the seamless transfer of assets from various EVM-compatible blockchains back to the Polygon ecosystem. This bridging solution allows developers to customize bridge plugins to meet specific requirements, facilitating interoperability between blockchains and different layers.

Unstoppable Networks

Supernets are pioneering unstoppable networks for the internet, where decentralized and secure applications can thrive.

bridge

The diagram above demonstrates how Supernets are interconnected with the Polygon PoS network, which benefits from the security properties of Ethereum. By leveraging blockchain technology, Supernets provide a strong foundation for building decentralized and blockchain-based solutions that can withstand adversarial conditions, resist censorship, and scale to meet the increasing demand for processing power, data storage, and transaction throughput.

Supernets employ a multi-faceted approach that leverages a combination of complementary scaling solutions to achieve maximum scalability. These solutions include layer-2 scaling techniques, parallelization, and, eventually, ZK technology.

bridge

By integrating these methods, Supernets can efficiently accommodate the increasing demand for processing power, data storage, and transaction throughput as the number of users and applications on the network grows.

Supernets Program

Implementation partners and service providers

To elaborate further, a Supernets program offers a range of service providers that can assist with various aspects of blockchain development, deployment, and maintenance. These include:

- **Node Providers**: These providers offer node infrastructure services that support the operation and maintenance of Supernets-based networks. Node providers can help ensure network stability, security, and scalability by providing reliable and optimized node infrastructure.
- **RPC Providers**: These providers offer remote procedure call (RPC) services that allow developers to interact with Supernet-based networks (childchains) and rootchain networks like the Mumbai PoS testnet and Polygon PoS mainnet. RPC providers can provide easy and reliable access to network data and functionality.
- **Smart Contract Monitoring**: This service enables real-time monitoring of smart contracts deployed on Supernets-based networks. This can help detect and respond to potential issues, vulnerabilities, or attacks on the network.
- **Oracle Services**: These services offer real-time data feeds and external information that can be used to support smart contract execution on Supernets-based networks. Oracles can help enable the development of more complex and advanced decentralized applications that rely on off-chain data and information.

- **Block Explorers**: These tools provide an interface for users to explore and visualize the blockchain's contents, including blocks, transactions, and other network data. Block explorers can help improve network transparency and accessibility and can be used by developers, users, and other stakeholders to gain insights into network activity and performance.
- **KYC Providers**: These providers offer Know Your Customer (KYC) and Anti-Money Laundering (AML) compliance services, which can help ensure that Supernets-based networks comply with regulatory requirements and mitigate risk for network participants.
- **Fiat On-Ramps**: These services bridge fiat currencies and digital assets, enabling users to purchase and use digital assets on Supernets-based networks. Fiat on-ramps can help increase accessibility and adoption of Supernets-based networks.

:::note Services will become available shortly

As partnerships develop and the ecosystem comes to life, new services will be ready for use. These services will become available shortly, and tutorials and other resources will be provided to help users make the most of them. Keep an eye out for updates as the ecosystem continues to grow and evolve.

:::

Easy deployments

Supernets provide hassle-free deployment for blockchain networks with Terraform scripts "one-click" deployments, allowing developers to seamlessly set up a childchain.

Cloud deployments

Supernets support cloud deployment options that enable developers and enterprises to easily and securely deploy a childchain to the cloud. With cloud deployment options, users can take advantage of the scalability and flexibility of cloud infrastructure, without having to worry about the complexities of managing their own infrastructure.

Supernets supports deployment to various cloud platforms and allow users to choose the cloud platform that best suits their needs and preferences.

Supernets cloud deployment options also come with a range of features and capabilities, such as auto-scaling, load balancing, and disaster recovery, that can help ensure network stability, security, and availability. These features can be especially important for enterprise-level deployments that require high levels of reliability and performance.

:::note Check out the local cloud deployment options

Get started with deploying a local private supernet on the cloud by checking out our cloud deployment guides available here.

:::

Why Supernets?

Regulatory Compliance and Network Sovereignty

Supernets offer network sovereignty and regulatory compliance by allowing maintainers to select network administrators that meet local regulations. This enables global networks to tailor their configurations to maintain compliance with regional regulatory requirements.

A New Approach to Infrastructure

Supernets offer a cutting-edge approach to system infrastructure, backed by an ecosystem of premium service providers. This empowers organizations with control over network properties and access while applying permissioned or permissionless infrastructure models. The integration of blockchain technology and industry-leading partners provides additional guarantees in security and efficiency, transforming traditional infrastructure designs and models.

Tokenization and Asset Management

Supernets facilitate the creation and management of utility tokens, incentivizing network participation and promoting efficient transactions. The ability to mint additional tokens provides flexibility to adjust token supply and distribution dynamically, ensuring network incentives remain aligned with user needs.

EVM Compatibility and Customization

Supernets focus on flexibility, adaptability, and EVM compatibility, streamlining the development process for developers. This design enables tailored blockchain solutions optimized for specific use cases while allowing the use of existing code and familiar programming languages. Developers can customize their networks with features such as default transaction fees and their own gas tokens, fine-tuning network performance and user experience.

Robust Consensus

Supernets utilize the PolyBFT consensus mechanism for exceptional throughput, minimal latency, and immediate network finality. This positions them as a highly suitable platform for a variety of use cases.

Cross-Chain Interoperability for Enhanced Efficiency

Supernets support cross-chain interoperability through their native bridge, enabling seamless asset transfers between different blockchains. This interconnected ecosystem allows developers to build on existing blockchain infrastructures and provides users with access to a broad range of applications and services. Cross-chain interoperability also facilitates increased collaboration and innovation between various blockchain networks.

On-chain Governance

Supernets enable on-chain governance, enabling participants to vote on changes to system parameters and promote transparency, accountability, and fairness. On-chain governance also allows for the creation of proto-jurisdictions to meet the needs of the community, defining rules and protocols for network operation, and promoting stability, security, innovation, and growth.

id: bls title: BLS Interface sidebar_label: BLS description: "Learn about the access control module and how to allowlist validators." keywords: - docs - polygon - edge - consensus - interface

The `IBLS` interface provides a set of functions for working with BLS (Boneh-Lynn-Shacham) signatures in a smart contract. This user guide will explain how to interact with the functions provided by the IBLS interface.

Functions

verifySingle()

This function verifies a single signature.

Parameters

- `signature` (uint256[2]): A 64-byte G1 group element (small signature).
- `pubkey` (uint256[4]): A 128-byte G2 group element (big pubkey).
- `message` (uint256[2]): The message signed to produce the signature.

Usage

To verify a single signature, call the verifySingle() function with the required parameters:

```
(bool sigVerification, bool callSuccess) = IBLS.instance.verifySingle(signature, pubkey, message);
```

verifyMultiple()

This function verifies multiple non-aggregated signatures where each message is unique.

Parameters

- `signature` (uint256[2]): A 64-byte G1 group element (small signature).
- `pubkeys` (uint256[4][]): An array of 128-byte G2 group elements (big pubkeys).
- `messages` (uint256[2][]): An array of messages signed to produce the signature.

Usage

To verify multiple non-aggregated signatures, call the verifyMultiple() function with the required parameters:

```
(bool checkResult, bool callSuccess) = IBLS.instance.verifyMultiple(signature, pubkeys, messages);
```

verifyMultipleSameMsg()

This function verifies an aggregated signature where the same message is signed.

Parameters

- `signature` (uint256[2]): A 64-byte G1 group element (small signature).
- `pubkeys` (uint256[4][]): An array of 128-byte G2 group elements (big pubkeys).
- `message` (uint256[2]): The message signed by all to produce the signature.

Usage

To verify an aggregated signature with the same message, call the verifyMultipleSameMsg() function with the required parameters:

```
(bool checkResult, bool callSuccess) = IBLS.instance.verifyMultipleSameMsg(signature, pubkeys, message);
```

mapToPoint()

This function maps a field element to the curve.

Parameters

`_x` (uint256): A valid field element.

Usage

To map a field element to the curve, call the mapToPoint() function with the required parameter:

```
uint256[2] memory point = IBLS.instance.mapToPoint(_x);
```

isValidSignature()

This function checks if a signature is formatted correctly and valid. It will revert if improperly formatted and return false if invalid.

Parameters

`signature` (uint256[2]): The BLS signature.

Usage

To check if a signature is valid, call the isValidSignature() function with the required parameter:

```
bool isValid = IBLS.instance.isValidSignature(signature);
```

isOnCurveG1()

This function checks if a point in the finite field Fq (x, y) is on the G1 curve.

Parameters

`point` (uint256[2]): An array with x and y values of the point.

Usage

To check if a point is on the G1 curve, call the isOnCurveG1() function with the required parameter:

```
bool isOnCurve = IBLS.instance.isOnCurveG1(point);
```

isOnCurveG2()

This function checks if a point in the finite field Fq (x, y) is on the G2 curve.

Parameters

---

id: bn256g2 title: BN256G2 Interface sidebar_label: BN256G2 description: "Learn about the access control module and how to allowlist validators." keywords: - docs - polygon - edge - consensus - interface

The `IBN256G2` interface provides functionality for performing addition operations on points in the G2 subgroup of the BN256 elliptic curve. This user guide will explain how to interact with the function provided by the `IBN256G2` interface.

Functions

ecTwistAdd()

This function performs the addition operation on two points in the G2 subgroup of the BN256 elliptic curve.

Parameters

- `pt1xx` (uint256): The xx-coordinate of the first point.
- `pt1xy` (uint256): The xy-coordinate of the first point.
- `pt1yx` (uint256): The yx-coordinate of the first point.
- `pt1yy` (uint256): The yy-coordinate of the first point.
- `pt2xx` (uint256): The xx-coordinate of the second point.
- `pt2xy` (uint256): The xy-coordinate of the second point.
- `pt2yx` (uint256): The yx-coordinate of the second point.
- `pt2yy` (uint256): The yy-coordinate of the second point.

Usage

To perform the addition operation on two points in the G2 subgroup of the BN256 elliptic curve, call the `ecTwistAdd()` function with the required parameters:

```
IBN256G2 bn256g2Instance = IBN256G2(bn256g2Address);

(uint256 pt3xx, uint256 pt3xy, uint256 pt3yx, uint256 pt3yy) = bn256g2Instance.ecTwistAdd(
    pt1xx, pt1xy, pt1yx, pt1yy,
    pt2xx, pt2xy, pt2yx, pt2yy
);
```

Replace `bn256g2Address` with the address of an existing `IBN256G2` implementation. Replace `pt1xx`, `pt1xy`, `pt1yx`, `pt1yy`, `pt2xx`, `pt2xy`, `pt2yx`, and `pt2yy` with the coordinates of the points you want to add. The resulting point, `pt3`, will have the coordinates (`pt3xx`, `pt3xy`, `pt3yx`, `pt3yy`).

The `ecTwistAdd()` function allows you to perform elliptic curve operations on the BN256 curve in the G2 subgroup, which can be useful for cryptographic applications such as zero-knowledge proofs, signature schemes, and more.

id: eip1559-interface title: EIP1559Burn sidebar_label: EIP1559Burn description: "Learn about the access control module and how to allowlist validators." keywords: - docs - polygon - edge - consensus - interface

The `EIP1559Burn` contract burns the native token on the rootchain as an ERC-20 token. This operation is crucial for maintaining the value of the token across different chains.

Functions

initialize()

```
function initialize(IChildERC20Predicate newChildERC20Predicate, address newBurnDestination)
```

This is the initialization function for the `EIP1559Burn` contract. It sets the `childERC20Predicate` and `burnDestination` state variables. This function can only be called once due to the initializer modifier.

Parameters:

- `newChildERC20Predicate` : Address of the ERC20 predicate on the childchain
- `newBurnDestination` : Address on the root chain to burn the tokens and send to

withdraw()

```
function withdraw() external {
        require(address(childERC20Predicate) != address(0), "EIP1559Burn: UNINITIALIZED");
        ...
}
```

This function burns the native tokens on the childchain and sends them to the burn destination on the rootchain. It requires the `childERC20Predicate` to be initialized, and takes the entire current native token balance and burns it.

Events

NativeTokenBurnt()

This event is emitted when native tokens are burnt. It includes the address of the burner and the amount of tokens that were burnt.

Usage

The `EIP1559Burn` contract is a part of the asset bridging process between the rootchain and the childchain. It interacts with the `IChildERC20Predicate` contract, which is responsible for handling the token transfer process.

In the asset bridging process, users deposit ERC20 tokens from the rootchain to the childchain, transact on the childchain, and then withdraw their assets back to the rootchain when they're done. The `withdraw()` function of the EIP1559Burn contract is called when users are ready to burn their tokens on the child chain and move them back to the rootchain.

Here is a high-level sequence of the bridging process:

- **Deposit**: Users deposit ERC20 tokens from the root chain to the childchain. The tokens are approved for transfer by the rootchain's ERC20 contract, and then deposited in the root chain's ERC20 predicate contract. The state of the deposit is synced to the childchain, and the tokens are minted on the child chain1
- **Withdraw**: When users want to move their assets back to the root chain, they initiate a withdrawal. The childchain's ERC20 predicate contract calls the `withdrawTo()` function, which burns the tokens on the childchain and syncs the state of the withdrawal back to the rootchain.
- **Exit**: The withdrawal is finalized on the root chain. The root chain's ERC20 predicate contract transfers the tokens back to the user.

---

id: childerc1155-predicate title: ChildERC1155 Predicate sidebar_label: ChildERC20 Predicate description: "Learn about the access control module and how to allowlist validators." keywords: - docs - polygon - edge - consensus - interface

The `IChildERC1155Predicate` interface is designed to work with the ERC1155-compliant token on the L2 side of the childchain. It handles the minting, burning, and state management for child tokens. This user guide will explain how to interact with the functions provided by the `IChildERC1155Predicate` interface.

Functions

initialize()

This function initializes the predicate with the required parameters.

Parameters

- `newL2StateSender` (address): The L2 state sender address.
- `newStateReceiver` (address): The state receiver address.
- `newRootERC721Predicate` (address): The root ERC721 predicate address.
- `newChildTokenTemplate` (address): The child token template address.

Usage

To initialize the `IChildERC1155Predicate` instance, call the `initialize()` function with the required parameters:

```
IChildERC1155Predicate.instance.initialize(newL2StateSender, newStateReceiver, newRootERC721Predicate, newChildTokenTemplate);
```

onStateReceive()

This function receives the state data from the main chain.

Parameters

- `id` (uint256): An identifier for the state data.
- `sender` (address): The sender address.
- `data` (bytes): The state data.

Usage

To handle state data received from the main chain, the predicate should call the `onStateReceive()` function with the required parameters:

```
IChildERC1155Predicate.instance.onStateReceive(id, sender, data);
```

withdraw()

This function allows the user to withdraw their child tokens to the main chain.

Parameters

- `childToken` (IChildERC1155): The child token contract.
- `tokenId` (uint256): The token identifier.
- `amount` (uint256): The amount of tokens to withdraw.

Usage

To withdraw child tokens to the main chain, call the `withdraw()` function with the required parameters:

```
IChildERC1155Predicate.instance.withdraw(childToken, tokenId, amount);
```

withdrawTo()

This function allows the user to withdraw their child tokens to a specific address on the main chain.

Parameters

- `childToken` (IChildERC1155): The child token contract.
- `receiver` (address): The address to receive the tokens on the main chain.
- `tokenId` (uint256): The token identifier.
- `amount` (uint256): The amount of tokens to withdraw.

Usage

To withdraw child tokens to a specific address on the main chain, call the `withdrawTo()` function with the required parameters:

```
IChildERC1155Predicate.instance.withdrawTo(childToken, receiver, tokenId, amount);
```

---

id: childerc1155 title: ChildERC1155 Interface sidebar_label: ChildERC1155 description: "Learn about the access control module and how to allowlist validators." keywords: - docs - polygon - edge - consensus - interface

The `IChildERC1155` interface represents an ERC1155-compliant token on the L2 side of the childchain. It allows the minting and burning of tokens under the control of a predicate address. This user guide will explain how to interact with the functions provided by the `IChildERC1155` interface.

Functions

initialize()

This function sets the values for the rootToken, name, and uri. The value for rootToken is immutable and can only be set once during initialization.

Parameters

- `rootToken_` (address): The address of the token on the rootchain.
- `name_` (string): The token's name.
- `uri_` (string): The token's metadata URI.

Usage

To initialize the IChildERC1155 instance, call the initialize() function with the required parameters:

```
IChildERC1155.instance.initialize(rootToken, name, uri);
```

predicate()

This function returns the predicate address controlling the child token.

Usage

To get the predicate address, call the predicate() function:

```
const predicateAddress = IChildERC1155.instance.predicate();
```

rootToken()

This function returns the address of the token on the rootchain.

Usage

To get the address of the token on the rootchain, call the rootToken() function:

```
const rootTokenAddress = IChildERC1155.instance.rootToken();
```

mint()

This function mints an NFT token to a particular address. The predicate address can only call it.

Parameters

- `account` (address): The user's account to mint the tokens.
- `id` (uint256): The index of NFT to mint to the account.
- `amount` (uint256): The amount of NFT to mint.

Usage

To mint an NFT token to an address, the predicate should call the mint() function with the required parameters:

```
bool success = IChildERC1155.instance.mint(account, id, amount);
```

mintBatch()

This function mints multiple NFTs to one address.

Parameters

- `accounts` (address[]): An array of addresses to mint each NFT to.
- `tokenIds` (uint256[]): An array of indexes of the NFTs to be minted.
- `amounts` (uint256[]): An array of the amount of each NFT to be minted.

Usage

To mint multiple NFTs to one address, call the mintBatch() function with the required parameters:

```
bool success = IChildERC1155.instance.mintBatch(accounts, tokenIds, amounts);
```

burn()

This function burns an NFT token from a particular address. The predicate address can only call it.

Parameters

- `from` (address): The user's account to burn the tokens from.
- `id` (uint256): The index of NFT to burn from the account.
- `amount` (uint256): The amount of NFT to burn.

Usage

To burn an NFT token from an address, the predicate should call the burn() function with the required parameters:

```
bool success = IChildERC1155.instance.burn(from, id, amount);
```

burnBatch()

This function burns multiple NFTs from one address.

Parameters

- `from` (address): The address to burn NFTs from.
- `tokenIds` (uint256[]): An array of indexes of the NFTs to be burned.
- `amounts` (uint256[]): An array of the amount of each N

Usage

To burn multiple NFTs from one address, call the burnBatch() function with the required parameters:

```
bool success = IChildERC1155.instance.burnBatch(from, tokenIds, amounts);
```

---

id: rooterc1155-predicate title: RootERC1155 Predicate sidebar_label: RootERC20 Predicate description: "Learn about the access control module and how to allowlist validators." keywords: - docs - polygon - edge - consensus - interface

The `IRootERC1155Predicate` interface is designed to work with ERC-1155 tokens on a rootchain. It provides functionality for depositing, withdrawing, and mapping ERC-1155 tokens between root and childchains. This user guide will explain how to interact with the functions provided by the

`IRootERC1155Predicate` interface.

Functions

deposit()

This function deposits tokens from the depositor to themselves on the childchain.

Parameters

- `rootToken` (IERC1155MetadataURI): The root token being deposited.
- `tokenId` (uint256): The index of the NFT to deposit.
- `amount` (uint256): The amount to deposit.

Usage

To deposit tokens from the depositor to themselves on the childchain, call the `deposit()` function with the required parameters:

```
IRootERC1155Predicate.instance.deposit(rootToken, tokenId, amount);
```

depositTo()

This function deposits tokens from the depositor to another address on the childchain.

Parameters

- `rootToken` (IERC1155MetadataURI): The root token being deposited.
- `receiver` (address): The address of the receiver on the childchain.
- `tokenId` (uint256): The index of the NFT to deposit.
- `amount` (uint256): The amount to deposit.

Usage

To deposit tokens from the depositor to another address on the childchain, call the `depositTo()` function with the required parameters:

```
IRootERC1155Predicate.instance.depositTo(rootToken, receiver, tokenId, amount);
```

depositBatch()

This function deposits tokens from the depositor to other addresses on the childchain.

Parameters

- `rootToken` (IERC1155MetadataURI): The root token being deposited.
- `receivers` (address[]): The addresses of the receivers on the childchain.
- `tokenIds` (uint256[]): The indices of the NFTs to deposit.
- `amounts` (uint256[]): The amounts to deposit.

Usage

To deposit tokens from the depositor to other addresses on the childchain, call the `depositBatch()` function with the required parameters:

```
IRootERC1155Predicate.instance.depositBatch(rootToken, receivers, tokenIds, amounts);
```

mapToken()

This function is used for token mapping.

Parameters

`rootToken` (IERC1155MetadataURI): The address of the root token to map.

Returns

`childToken` (address): The address of the mapped child token.

Usage

To map a token, call the mapToken() function with the required parameter:

```
address childToken = IRootERC1155Predicate.instance.mapToken(rootToken);
```

This function is called internally on deposit if the token is not mapped already.

id: childerc20-predicate title: ChildERC20 Predicate sidebar_label: ChildERC20 Predicate description: "Learn about the access control module and how to allowlist validators." keywords: - docs - polygon - edge - consensus - interface

The `IChildERC20Predicate` interface is part of the cross-chain bridge system, which manages the child tokens on the L2 side. This guide will explain the methods provided by the IChildERC20Predicate interface and how to use them.

Overview

The `IChildERC20Predicate` interface is responsible for managing child tokens on L2. It provides:

- Methods for deploying new child tokens.
- Processing state updates received from L1.
- Withdrawing tokens back to L1.

Initialize

Before using the IChildERC20Predicate contract, it must be initialized with the required parameters:

```
function initialize(
    address newL2StateSender,
    address newStateReceiver,
    address newRootERC20Predicate,
    address newChildTokenTemplate,
    address newNativeTokenRootAddress,
    string calldata newNativeTokenName,
    string calldata newNativeTokenSymbol,
    uint8 newNativeTokenDecimals
) external;
```

The parameters for initialization include the addresses of the L2 state sender, the state receiver, the root ERC20 predicate, and the child token template. The initialization also requires the native token's root address, name, symbol, and decimals.

Deploy a new child token

To create a new child token on the L2 side, call the deployChildToken method:

```
function deployChildToken(
    address rootToken,
    bytes32 salt,
    string calldata name,
    string calldata symbol,
    uint8 decimals
) external;
```

The parameters for deploying a new child token include the address of the corresponding root token on L1, a unique salt value, and the token's name, symbol, and decimals.

Process state updates

The onStateReceive method receives state updates from the L1 side and processes the data accordingly:

```
function onStateReceive(uint256 /* id */, address sender, bytes calldata data) external;
```

When a state update is received, the method processes the data based on the sender's address and provided data. The system typically calls this method automatically when state updates are received from L1.

Withdraw tokens

To withdraw tokens from L2 back to L1, call the withdraw method:

```
function withdraw(IChildERC20 childToken, uint256 amount) external;
```

This method takes the child token contract address and the amount to be withdrawn. The tokens will be withdrawn to the message sender's address on L1.

Withdraw tokens to a specific address

If you want to withdraw tokens from L2 to L1 for a specific address, use the `withdrawTo` method:

```
function withdrawTo(IChildERC20 childToken, address receiver, uint256 amount) external;
```

This method takes the child token contract address, the receiver's address, and the amount to be withdrawn. The tokens will be withdrawn to the specified receiver's address on L1.

id: childerc20 title: ChildERC20 Interface sidebar_label: ChildERC20 description: "Learn about the access control module and how to allowlist validators." keywords: - docs - polygon - edge - consensus - interface

The `IChildERC20` interface extends the `IERC20MetadataUpgradeable` interface to provide additional token functionality on a childchain or layer 2 solution. This guide explains how to interact with the interface and its methods.

Overview

The `IChildERC20` interface includes methods for:

- Initializing the token
- Retrieving the predicate and root token addresses
- Minting and burning tokens

Initializing the Token

To initialize a child token, call the initialize function with the following parameters:

- `rootToken_` : The address of the root token on the main chain.
- `name_` : The name of the token.
- `symbol_` : The symbol of the token.
- `decimals_` : The number of decimals used to define the smallest unit of the token.

Example

```
childToken.initialize(rootTokenAddress, "My Token", "MTK", 18);
```

Retrieving the Predicate and Root Token Addresses

To get the predicate and root token addresses, call the `predicate` and `rootToken` functions respectively.

Example:

```
address predicateAddress = childToken.predicate();
address rootTokenAddress = childToken.rootToken();
```

Minting and Burning Tokens

The `mint` and `burn` functions can only be called by the predicate address, which typically represents a bridge contract responsible for managing the token's supply between the main chain and the childchain.

Minting Tokens

To mint tokens, call the mint function with the following parameters:

- `account` : The user's address to mint the tokens.
- `amount` : The amount of tokens to mint to the account.

```
bool success = childToken.mint(userAddress, 100 * 10**18);
```

Burning Tokens

To burn tokens, call the burn function with the following parameters:

- `account` : The user's address to burn the tokens.
- `amount` : The amount of tokens to burn from the account.

Example:

```
bool success = childToken.burn(userAddress, 50 * 10**18);
```

> *Note: The predicate address should only call the mint and burn functions. If you're a developer integrating with a child token, you typically won't need to interact with these functions directly. Instead, you'll interact with the bridge or the main chain's root token contract to move tokens between chains.*

---

id: native-erc20 title: NativeERC20 sidebar_label: NativeERC20 description: "Learn about the access control module and how to allowlist validators." keywords: - docs - polygon - edge - consensus - interface

The `INativeERC20` interface represents a native ERC20 token on the L2 side of the childchain. It allows the minting and burning of tokens under the control of a predicate address. This user guide will explain how to interact with the functions provided by the `INativeERC20` interface.

Functions

initialize()

This function sets the values for the predicate, rootToken, name, symbol, and decimals.

Parameters

- `predicate_` (address): The address of the predicate controlling the child token.
- `rootToken_` (address): The address of the root token on the mainchain.
- `name_` (string): The token's name.

- `symbol_` (string): The token's symbol.
- `decimals_` (uint8): The number of decimals for the token.

Usage

To initialize the INativeERC20 instance, call the `initialize()` function with the required parameters:

```
INativeERC20.instance.initialize(predicate, rootToken, name, symbol, decimals);
```

predicate()

This function returns the predicate address controlling the child token.

Usage

To get the predicate address, call the `predicate()` function:

```
const predicateAddress = INativeERC20.instance.predicate();
```

rootToken()

This function returns the root token address.

Usage

To get the root token address, call the `rootToken()` function:

```
const rootTokenAddress = INativeERC20.instance.rootToken();
```

mint()

This function mints an amount of tokens to a particular address. The predicate address can only call it.

Parameters

- `account` (address): The user's account to mint the tokens.
- `amount` (uint256): The token amount to mint to the account.

Usage

To mint tokens to an address, the predicate should call the `mint()` function with the required parameters:

```
INativeERC20.instance.mint(account, amount);
```

burn()

This function burns an amount of tokens from a particular address. The predicate address can only call it.

Parameters

- `account` (address): The user's account to burn the tokens from.
- `amount` (uint256): The amount of tokens to burn from the account.

Usage

To burn tokens from an address, the predicate should call the `burn()` function with the required parameters:

```
INativeERC20.instance.burn(account, amount);
```

These functions enable interaction with native ERC20 tokens on the L2 side of the childchain. Developers can use these functions to mint, burn, and retrieve information about native ERC20 tokens.

id: rooterc20-predicate title: RootERC20 Predicate sidebar_label: RootERC20 Predicate description: "Learn about the access control module and how to allowlist validators." keywords: - docs - polygon - edge - consensus - interface

The `IRootERC20Predicate` interface is designed to work with ERC20 tokens on a rootchain. It provides functionality for depositing and mapping ERC20 tokens between root and childchains. This user guide will explain how to interact with the functions provided by the `IRootERC20Predicate` interface.

Functions

deposit()

This function deposits tokens from the depositor to themselves on the childchain.

Parameters

- `rootToken` (IERC20Metadata): The root token being deposited.

- `amount` (uint256): The amount to deposit.

Usage

To deposit tokens from the depositor to themselves on the childchain, call the `deposit()` function with the required parameters:

```
IRootERC20Predicate.instance.deposit(rootToken, amount);
```

depositTo()

This function deposits tokens from the depositor to another address on the childchain.

Parameters

- `rootToken` (IERC20Metadata): The root token being deposited.
- `receiver` (address): The address of the receiver on the childchain.
- `amount` (uint256): The amount to deposit.

Usage

To deposit tokens from the depositor to another address on the childchain, call the `depositTo()` function with the required parameters:

```
IRootERC20Predicate.instance.depositTo(rootToken, receiver, amount);
```

mapToken()

This function is used for token mapping.

Parameters

rootToken (IERC20Metadata): The address of the root token to map.

Usage

The mapToken() function is called internally on deposit if the token is not mapped already.

id: childerc721-predicate title: ChildERC721 Predicate sidebar_label: ChildERC721 Predicate description: "Learn about the access control module and how to allowlist validators." keywords: - docs - polygon - edge - consensus - interface

The `IChildERC721Predicate` interface is designed to work in conjunction with the `IChildERC721` interface to enable the withdrawal of ERC721-compliant NFT tokens from a childchain back to the rootchain. This user guide will explain how to interact with the functions provided by the `IChildERC721Predicate` interface.

Functions

initialize()

This function initializes the predicate contract with required parameters.

Parameters

- `newL2StateSender` (address): The address of the L2 state sender.
- `newStateReceiver` (address): The address of the state receiver.
- `newRootERC721Predicate` (address): The address of the root ERC721 predicate.
- `newChildTokenTemplate` (address): The address of the child token template.

Usage

To initialize the `IChildERC721Predicate` instance, call the `initialize()` function with the required parameters:

```
IChildERC721Predicate.instance.initialize(newL2StateSender, newStateReceiver, newRootERC721Predicate, newChildTokenTemplate);
```

onStateReceive()

This function is called when the predicate receives a state update from the L2 state sender.

Parameters

- `id` (uint256): An identifier, not used in this implementation.
- `sender` (address): The address of the sender.
- `data` (bytes): The calldata to be processed.

Usage

The `onStateReceive()` function is not intended to be called directly by developers. It is automatically called when the predicate receives a state update from the L2 state sender.

withdraw()

This function withdraws an NFT token to the original owner.

Parameters

- `childToken` (IChildERC721): The child token contract instance.
- `tokenId` (uint256): The token identifier.

Usage

To withdraw an NFT token to the original owner, call the `withdraw()` function with the required parameters:

```
IChildERC721Predicate.instance.withdraw(childToken, tokenId);
```

withdrawTo()

This function withdraws an NFT token to a specified address.

Parameters

- `childToken` (IChildERC721): The child token contract instance.
- `receiver` (address): The address of the recipient.
- `tokenId` (uint256): The token identifier.

Usage

To withdraw an NFT token to a specified address, call the `withdrawTo()` function with the required parameters:

```
IChildERC721Predicate.instance.withdrawTo(childToken, receiver, tokenId);
```

withdrawBatch()

This function withdraws multiple NFT tokens to specified addresses in one transaction.

Parameters

- `childToken` (IChildERC721): The child token contract instance.
- `receivers` (address[]): An array of recipient addresses.
- `tokenIds` (uint256[]): An array of token identifiers.

Usage

To withdraw multiple NFT tokens to specified addresses in one transaction, call the `withdrawBatch()` function with the required parameters:

```
IChildERC721Predicate.instance.withdrawBatch(childToken, receivers, tokenIds);
```

---

id: childerc721 title: ChildERC721 Interface sidebar_label: ChildERC721 description: "Learn about the access control module and how to allowlist validators." keywords: - docs - polygon - edge - consensus - interface

The `IChildERC721` interface is designed to work with ERC721-compliant NFT tokens on a childchain. It provides functionality for minting, burning, and managing NFTs. This user guide will explain how to interact with the functions provided by the `IChildERC721` interface.

Functions

initialize()

This function initializes the child token contract with the root token address, name, and symbol.

Parameters

- `rootToken_` (address): The root token address.
- `name_` (string): The name of the token.
- `symbol_` (string): The symbol of the token.

Usage

To initialize the IChildERC721 instance, call the `initialize()` function with the required parameters:

```
IChildERC721.instance.initialize(rootToken_, name_, symbol_);
```

predicate()

This function returns the predicate address controlling the child token.

Usage

To get the predicate address, call the predicate() function:

```
address predicateAddress = IChildERC721.instance.predicate();
```

rootToken()

This function returns the address of the token on the rootchain.

Usage

To get the root token address, call the `rootToken()` function:

```
address rootTokenAddress = IChildERC721.instance.rootToken();
```

mint()

This function mints an NFT token to a specific address.

Parameters

- `account` (address): The recipient address.
- `tokenId` (uint256): The token identifier.

Usage

To mint an NFT token to a specific address, call the `mint()` function with the required parameters:

```
bool success = IChildERC721.instance.mint(account, tokenId);
```

mintBatch()

This function mints multiple NFT tokens in one transaction.

Parameters

- `accounts` (address[]): An array of recipient addresses.
- `tokenIds` (uint256[]): An array of token identifiers.

Usage

To mint multiple NFT tokens in one transaction, call the `mintBatch()` function with the required parameters:

```
bool success = IChildERC721.instance.mintBatch(accounts, tokenIds);
```

burn()

This function burns an NFT token from a specific address.

Parameters

- `account` (address): The address to burn the NFT from.
- `tokenId` (uint256): The token identifier.

Usage

To burn an NFT token from a specific address, call the `burn()` function with the required parameters:

```
bool success = IChildERC721.instance.burn(account, tokenId);
```

burnBatch()

This function burns multiple NFT tokens from a specific address in one transaction.

Parameters

- `account` (address): The address to burn the NFTs from.
- `tokenIds` (uint256[]): An array of token identifiers.

Usage

To burn multiple NFT tokens from a specific address in one transaction, call the `burnBatch()` function with the required parameters:

```
bool success = IChildERC721.instance.burnBatch(account, tokenIds);
```

---

id: rooterc721-predicate title: RootERC721 Predicate sidebar_label: RootERC721 Predicate description: "Learn about the access control module and how to allowlist validators." keywords: - docs - polygon - edge - consensus - interface

The `IRootERC721Predicate` interface is designed to work with ERC721-compliant NFT tokens on a rootchain. It provides functionality for depositing and mapping ERC721 tokens between root and childchains. This user guide will explain how to interact with the functions provided by the `IRootERC721Predicate` interface.

Functions

deposit()

This function deposits tokens from the depositor to themselves on the childchain.

Parameters

- `rootToken` (IERC721Metadata): The root token being deposited.
- `tokenId` (uint256): The index of the NFT to deposit.

Usage

To deposit tokens from the depositor to themselves on the childchain, call the `deposit()` function with the required parameters:

```
IRootERC721Predicate.instance.deposit(rootToken, tokenId);
```

depositTo()

This function deposits tokens from the depositor to another address on the childchain.

Parameters

- `rootToken` (IERC721Metadata): The root token being deposited.
- `receiver` (address): The address of the receiver on the childchain.
- `tokenId` (uint256): The index of the NFT to deposit.

Usage

To deposit tokens from the depositor to another address on the childchain, call the depositTo() function with the required parameters:

```
IRootERC721Predicate.instance.depositTo(rootToken, receiver, tokenId);
```

depositBatch()

This function deposits tokens from the depositor to other addresses on the childchain.

Parameters

- `rootToken` (IERC721Metadata): The root token being deposited.
- `receivers` (address[]): An array of addresses of the receivers on the childchain.
- `tokenIds` (uint256[]): An array of indices of the NFTs to deposit.

Usage

To deposit tokens from the depositor to other addresses on the childchain, call the `depositBatch()` function with the required parameters:

```
IRootERC721Predicate.instance.depositBatch(rootToken, receivers, tokenIds);
```

mapToken()

This function is used for token mapping.

Parameters

`rootToken` (IERC721Metadata): The address of the root token to map.

Usage

The `mapToken()` function is called internally on deposit if the token is not mapped already.

id: checkpoint-manager title: CheckpointManager sidebar_label: CheckpointManager description: "Learn about the access control module and how to allowlist validators." keywords: - docs - polygon - edge - consensus - interface

The `ICheckpointManager` interface is a crucial part of the checkpoint system, allowing validators to submit signed checkpoints as proof of the canonical chain. This guide will explain the methods the ICheckpointManager interface provides and how to use them.

Overview

The `ICheckpointManager` interface is responsible for managing checkpoints in the system. It provides methods for submitting checkpoints, verifying event membership by block number or epoch, and retrieving checkpoint information.

Submit a checkpoint

To submit a checkpoint, call the submit method:

```
function submit(
    CheckpointMetadata calldata checkpointMetadata,
    Checkpoint calldata checkpoint,
    uint256[2] calldata signature,
    Validator[] calldata newValidatorSet,
```

```
    bytes calldata bitmap
) external;
```

The parameters for submitting a checkpoint include the following:

- The checkpoint metadata.
- The checkpoint itself.
- The aggregated signature.
- The new validator set.
- A bitmap of the old validator set that signed the message.
- The contract will verify the provided signature against the stored validator set.

Get event membership by block number

To check if an event is part of the event root for a given block number, call the getEventMembershipByBlockNumber method:

```
function getEventMembershipByBlockNumber(
    uint256 blockNumber,
    bytes32 leaf,
    uint256 leafIndex,
    bytes32[] calldata proof
) external view returns (bool);
```

This method takes the block number, the leaf of the event (keccak256-encoded log), the leaf index in the Merkle root tree, and the proof for leaf membership in the event root tree. It returns true if the event is part of the event root.

Get event membership by epoch

To check if an event is part of the event root for a given epoch, call the getEventMembershipByEpoch method:

```
function getEventMembershipByEpoch(
    uint256 epoch,
    bytes32 leaf,
    uint256 leafIndex,
    bytes32[] calldata proof
) external view returns (bool);
```

This method takes the epoch ID, the leaf of the event (keccak256-encoded log), the leaf index in the Merkle root tree, and the proof for leaf membership in the event root tree. It returns true if the event is part of the event root.

Get checkpoint block number

To get the checkpoint block number for a given block number, call the getCheckpointBlock method:

```
function getCheckpointBlock(uint256 blockNumber) external view returns (bool, uint256);
```

This method takes the block number and returns a boolean indicating if the block number was checkpointed, as well as the checkpoint block number.

Get event root by block

To get the event root for a given block number, call the getEventRootByBlock method:

```
function getEventRootByBlock(uint256 blockNumber) external view returns (bytes32);
```

---

id: exit-helper title: ExitHelper sidebar_label: ExitHelper description: "Learn about the access control module and how to allowlist validators." keywords: - docs - polygon - edge - consensus - interface

The `IExitHelper` interface is a helper contract that processes exits from stored event roots in `CheckpointManager`. It allows users to perform an exit for one or multiple events. This user guide will explain how to interact with the functions provided by the IExitHelper interface.

Functions

exit()

This function allows you to perform an exit for one event.

Parameters

- `blockNumber` (uint256): The block number of the exit event on L2.
- `leafIndex` (uint256): The leaf index in the exit event Merkle tree.
- `unhashedLeaf` (bytes): The ABI-encoded exit event leaf.
- `proof` (bytes32[]): The proof of the event inclusion in the tree.

Usage

To exit an event, call the `exit()` function with the required parameters:

```
IExitHelper.exitHelperInstance.exit(blockNumber, leafIndex, unhashedLeaf, proof);
```

batchExit()

This function allows you to perform a batch exit for multiple events.

Parameters

- `inputs` (BatchExitInput[]): An array of BatchExitInput structs, where each struct contains the following fields:
- `blockNumber` (uint256): The block number of the exit event on L2.
- `leafIndex` (uint256): The leaf index in the exit event Merkle tree.
- `unhashedLeaf` (bytes): The ABI-encoded exit event leaf.
- `proof` (bytes32[]): The proof of the event inclusion in the tree.

Usage

To perform a batch exit for multiple events, create an array of `BatchExitInput` structs and call the `batchExit()` function:

```
let batchExitInputs = [
  {
    blockNumber: blockNumber1,
    leafIndex: leafIndex1,
    unhashedLeaf: unhashedLeaf1,
    proof: proof1
  },
  {
    blockNumber: blockNumber2,
    leafIndex: leafIndex2,
    unhashedLeaf: unhashedLeaf2,
    proof: proof2
  }
];

IExitHelper.exitHelperInstance.batchExit(batchExitInputs);
```

This will process exits for multiple events in a single transaction, which can save gas and improve efficiency.

id: state-receiver title: StateReceiver sidebar_label: StateReceiver description: "Learn about the access control module and how to allowlist validators." keywords: - docs - polygon - edge - consensus - interface

The `IStateReceiver` interface is designed to handle state updates received from a higher-level entity, typically on the L2 side of the childchain. This user guide will explain how to interact with the functions provided by the `IStateReceiver` interface.

Functions

onStateReceive()

This function is called when the contract receives a state update. It processes the incoming state, allowing the contract to react to changes in the higher-level state.

Parameters

- `counter` (uint256): A counter value associated with the state update.
- `sender` (address): The entity's address sending the state update.
- `data` (bytes): The calldata associated with the state update.

Usage

To handle a state update in your contract, you must implement the onStateReceive() function. The higher-level entity will call this function when a state update is sent.

For example, you might implement onStateReceive() as follows:

```
contract MyStateReceiver is IStateReceiver {
    function onStateReceive(uint256 counter, address sender, bytes calldata data) external override {
        // Process the incoming state update
        // ...
    }
}
```

This function allows your contract to react to state updates, enabling communication between your contract and higher-level state providers. By implementing the `IStateReceiver` interface, you can

ensure that your contract stays current with broader ecosystem changes.

id: state-sender title: StateSender sidebar_label: StateSender description: "Learn about the access control module and how to allowlist validators." keywords: - docs - polygon - edge - consensus - interface

The `IStateSender` interface provides a way to send state updates from one contract to another, often between different layers of a network, such as L1 and L2. This user guide will help you understand how to interact with the functions provided by the `IStateSender` interface.

Functions

syncState()

This function sends a state update to a specified contract, allowing the receiving contract to process the changes in the state.

Parameters

- `receiver` (address): The contract address will receive the state update.
- `data` (bytes): The calldata associated with the state update to be sent.

Usage

You'll need to call the syncState() function to send a state update from your contract. This function will send the state update to the specified receiver contract.

For example, you might use the `syncState()` function as follows:

```
contract MyStateSender is IStateSender {
    function sendStateUpdate(address receiver, bytes calldata data) public {
        // Send the state update to the specified receiver contract
        syncState(receiver, data);
    }
}
```

This function enables your contract to communicate state updates to other contracts, facilitating cross-contract and cross-layer communication. By implementing the IStateSender interface, you can ensure your contract can send state updates to other parts of the ecosystem.

id: custom-supernet-manager-interface title: CustomSupernetManager sidebar_label: CustomSupernetManager description: "Learn about the access control module and how to allowlist validators." keywords: - docs - polygon - edge - consensus - interface

The `CustomSupernetManager` contract manages validator access and syncs voting power between the stake manager and the validator set on the child chain. It implements the base `SupernetManager` contract.

Events

AddedToWhitelist()

```
event AddedToWhitelist(address indexed validator);
```

This event is emitted when a validator is added to the whitelist of validators allowed to stake.

RemovedFromWhitelist()

```
event RemovedFromWhitelist(address indexed validator);
```

This event is emitted when a validator is removed from the whitelist of validators allowed to stake.

ValidatorRegistered()

```
event ValidatorRegistered(address indexed validator, uint256[4] blsKey);
```

This event is emitted when a validator is registered with their public key.

ValidatorDeactivated()

```
event ValidatorDeactivated(address validator);
```

This event is emitted when a validator is deactivated.

GenesisFinalized()

```
event GenesisFinalized(uint256 amountValidators);
```

This event is emitted when the initial genesis validator set is finalized.

StakingEnabled()

```
event StakingEnabled();
```

This event is emitted when staking is enabled after the successful initialization of the child chain.

Functions

whitelistValidators()

```
function whitelistValidators(address[] calldata validators_) external;
```

This function allows whitelisting validators that are allowed to stake. It can only be called by the owner of the contract.

Parameters:

`validators_` : An array of addresses representing the validators to be whitelisted.

register()

```
function register(uint256[2] calldata signature, uint256[4] calldata pubkey) external;
```

This function is called to register the public key of a validator. It validates the signature and registers the validator with their public key.

Parameters:

- `signature` : An array of two 256-bit integers representing the signature.
- `pubkey` : An array of four 256-bit integers representing the public key of the validator.

finalizeGenesis()

```
function finalizeGenesis() external;
```

This function is called to finalize the initial genesis validator set. It can only be called by the owner of the contract.

enableStaking()

```
function enableStaking() external;
```

This function is called to enable staking after the successful initialization of the child chain. It can only be called by the owner of the contract.

withdrawSlashedStake()

```
function withdrawSlashedStake(address to) external;
```

This function is called to withdraw slashed MATIC of slashed validators. It can only be called by the owner of the contract.

Parameters:

`to` : The address where the slashed stake will be withdrawn.

onL2StateReceive()

```
function onL2StateReceive(uint256 /*id*/, address sender, bytes calldata data) external;
```

This function is called by the exit helpers to either release the stake of a validator or slash it. It can only be called after the genesis and must be synced from the child chain.

Parameters:

- `sender` : The address of the sender calling the function.
- `data` : The data containing the information about the stake release or slash.

genesisSet()

```
function genesisSet() external view returns (GenesisValidator[] memory);
```

This function returns the initial genesis validator set with their balances.

Returns:

`GenesisValidator[]` : An array of GenesisValidator structs representing the initial genesis validator set.

getValidator()

```
function getValidator(address validator_) external view returns (Validator memory);
```

This function returns the Validator instance based on the provided validator address.

Parameters:

`validator_` : The address of the validator.

Returns:

`Validator` : The Validator struct representing the validator instance.

id: stake-manager-interface title: StakeManager sidebar_label: StakeManager description: "Learn about the access control module and how to allowlist validators." keywords: - docs - polygon - edge - consensus - interface

The `StakeManager` interface manages stakes for all childchains in the Supernets architecture.

Events

ChildManagerRegistered()

```
event ChildManagerRegistered(uint256 indexed id, address indexed manager);
```

This event is emitted when a new child chain manager is registered with the StakeManager contract. It includes the ID of the child chain and the address of the manager contract.

StakeAdded()

```
event StakeAdded(uint256 indexed id, address indexed validator, uint256 amount);
```

This event is emitted when a validator adds stake for a child chain. It includes the ID of the child chain, the address of the validator, and the amount of stake added.

StakeRemoved()

```
event StakeRemoved(uint256 indexed id, address indexed validator, uint256 amount);
```

This event is emitted when a validator's stake is removed for a child chain. It includes the ID of the child chain, the address of the validator, and the amount of stake removed.

StakeWithdrawn()

```
event StakeWithdrawn(address indexed validator, address indexed recipient, uint256 amount);
```

This event is emitted when a validator withdraws their released stake. It includes the address of the validator, the address of the recipient, and the amount of stake withdrawn.

ValidatorSlashed()

```
event ValidatorSlashed(uint256 indexed id, address indexed validator, uint256 amount);
```

This event is emitted when a validator's stake is slashed for a child chain. It includes the ID of the child chain, the address of the validator, and the amount of stake slashed.

Functions

registerChildChain()

```
function registerChildChain(address manager) external returns (uint256 id);
```

This function is called to register a new child chain with the StakeManager contract. It associates the child chain with its manager contract.

Parameters:

`manager` : The address of the manager contract for the child chain.

stakeFor()

```
function releaseStakeOf(address validator, uint256 amount) external;
```

This function is called by a validator to stake for a child chain. It adds the specified amount of stake for the given child chain.

Parameters:

- `id` : The ID of the child chain.
- `amount` : The amount of stake to be added.

releaseStakeOf()

```
function releaseStakeOf(address validator, uint256 amount) external;
```

This function is called by the child manager contract to release a validator's stake. It releases the specified amount of stake for the given validator.

Parameters:

- `validator` : The address of the validator.
- `amount` : The amount of stake to be released.

withdrawStake()

```
function withdrawStake(address to, uint256 amount) external;
```

This function allows a validator to withdraw their released stake. It transfers the specified amount of released stake to the specified recipient address.

Parameters:

- `to` : The address where the withdrawn stake will be sent.
- `amount` : The amount of released stake to be withdrawn.

slashStakeOf()

```
function slashStakeOf(address validator, uint256 amount) external;
```

This function is called by the child manager contract to slash a validator's stake. It slashes the specified amount of stake for the given validator, and the manager contract collects the slashed amount.

Parameters:

- `validator` : The address of the validator.
- `amount` : The amount of stake to be slashed.

withdrawableStake()

```
function withdrawableStake(address validator) external view returns (uint256 amount);
```

This function returns the amount of stake that a validator can withdraw. It represents the released stake that is available for withdrawal.

Parameters:

`validator` : The address of the validator.

Returns:

`amount` : The amount of stake that the validator can withdraw.

totalStake()

```
function totalStake() external view returns (uint256 amount);
```

This function returns the total amount of stake staked for all child chains.

Returns:

`amount` : The total amount of stake staked for all child chains.

totalStakeOfChild()

```
function totalStakeOfChild(uint256 id) external view returns (uint256 amount);
```

This function returns the total amount of stake staked for a specific child chain.

Parameters:

`id` : The ID of the child chain.

Returns:

`amount` : The total amount of stake staked for the specified child chain.

totalStakeOf()

```
function totalStakeOf(address validator) external view returns (uint256 amount);
```

This function returns the total amount of stake staked by a validator for all child chains.

Parameters:

`validator` : The address of the validator.

Returns:

`amount` : The total amount of stake staked by the specified validator.

stakeOf()

```
function stakeOf(address validator, uint256 id) external view returns (uint256 amount);
```

This function returns the amount of stake staked by a validator for a specific child chain.

Parameters:

- `validator` : The address of the validator.
- `id` : The ID of the child chain.

Returns:

`amount` : The amount of stake staked by the specified validator for the specified child chain.

managerOf()

```
function managerOf(uint256 id) external view returns (ISupernetManager manager);
```

This function returns the child chain manager contract for a specific child chain.

Parameters:

`id` : The ID of the child chain.

Returns:

`manager` : The address of the child chain manager contract.

idFor()

```
function idFor(address manager) external view returns (uint256 id);
```

This function returns the child chain ID associated with a specific child chain manager contract.

Parameters:

`manager` : The address of the child chain manager contract.

Returns:

`id` : The ID of the child chain associated with the specified manager contract.

id: supernet-manager-interface title: SupernetManager sidebar_label: SupernetManager description: "Learn about the access control module and how to allowlist validators." keywords: - docs - polygon - edge - consensus - interface

The `SupernetManager` contract is an abstract contract for managing Supernets. It serves as a base contract that should be implemented with custom desired functionality.

Functions

onInit()

```
function onInit(uint256 id) external;
```

This function is called when a new childchain is registered. It allows the implementing contract to perform initialization logic when a new child chain is added to the Supernet.

Parameters:

`id` : The ID of the newly registered child chain.

onStake()

```
function onStake(address validator, uint256 amount) external;
```

This function is called when a validator stakes. It allows the implementing contract to perform additional logic when a validator stakes for a child chain in the supernet.

Parameters:

- `validator` : The address of the validator who stakes.
- `amount` : The amount of stake being staked by the validator.

---

id: root-validator-set title: RootValidatorSet Interface sidebar_label: RootValidatorSet description: "Learn about the access control module and how to allowlist validators." keywords: - docs - polygon - edge - consensus - interface

The `IRootValidatorSet` interface provides functionality for managing the validator set on the rootchain. It allows adding new validators and querying validator information. This user guide will explain how to interact with the functions provided by the `IRootValidatorSet` interface.

Data Structures

Validator

The Validator struct represents a validator in the validator set.

- `_address` (address): The Ethereum address of the validator.
- `blsKey` (uint256[4]): The BLS public key of the validator.

Functions

addValidators()

This function adds new validators to the validator set.

Parameters

`newValidators` (Validator[]): An array of Validator structs representing the new validators to be added.

Usage

To add new validators to the validator set, call the `addValidators()` function with the required parameters:

```
IRootValidatorSet rootValidatorSetInstance = IRootValidatorSet(rootValidatorSetAddress);

IRootValidatorSet.Validator[] memory newValidators = new IRootValidatorSet.Validator[](2);

newValidators[0] = IRootValidatorSet.Validator(validatorAddress1, blsKey1);
newValidators[1] = IRootValidatorSet.Validator(validatorAddress2, blsKey2);

rootValidatorSetInstance.addValidators(newValidators);
```

Replace `rootValidatorSetAddress` with the address of an existing `IRootValidatorSet` implementation. Replace `validatorAddress1` , `blsKey1` , `validatorAddress2` , and `blsKey2` with the Ethereum addresses and BLS public keys of the new validators.

getValidatorBlsKey()

This function returns the BLS public key of a validator by their ID.

Parameters

`id` (uint256): The ID of the validator.

Usage

To get the BLS public key of a validator by their ID, call the `getValidatorBlsKey()` function with the required parameter:

```
IRootValidatorSet rootValidatorSetInstance = IRootValidatorSet(rootValidatorSetAddress);

uint256[4] memory blsKey = rootValidatorSetInstance.getValidatorBlsKey(validatorId);
```

Replace rootValidatorSetAddress with the address of an existing `IRootValidatorSet` implementation. Replace `validatorId` with the ID of the validator whose BLS public key you want to retrieve.

activeValidatorSetSize()

This function returns the size of the active validator set.

Usage

To get the size of the active validator set, call the `activeValidatorSetSize()` function:

```
IRootValidatorSet rootValidatorSetInstance = IRootValidatorSet(rootValidatorSetAddress);

uint256 validatorSetSize = rootValidatorSetInstance.activeValidatorSetSize();
```

Replace rootValidatorSetAddress with the address of an existing `IRootValidatorSet` implementation. The function will return the number of active validators in the validator set.

id: validator-set-base title: ValidatorSetBase Interface sidebar_label: ValidatorSetBase description: "Learn about the access control module and how to allowlist validators." keywords: - docs - polygon - edge - consensus - interface

The `IChildValidatorSetBase` interface is part of the childchain. It handles validator registration, stake storage, and reward distribution. This user guide will explain the key components and functions of the interface.

Structs

InitStruct

InitStruct is used during the initialization of the contract.

- `epochReward` (uint256): The reward per epoch.
- `minStake` (uint256): The minimum amount a validator must stake.
- `minDelegation` (uint256): The minimum amount for delegating MATIC tokens.
- `epochSize` (uint256): The size of an epoch.

ValidatorInit

ValidatorInit is used when initializing a validator.

- `addr` (address): The validator's address.
- `pubkey` (uint256[4]): The validator's public BLS key.
- `signature` (uint256[2]): The validator's signature.
- `stake` (uint256): The amount staked by the validator.

DoubleSignerSlashingInput

DoubleSignerSlashingInput represents the information about double signers to be slashed along with signatures and bitmap.

- `epochId` (uint256): The ID of the epoch.
- `eventRoot` (bytes32): The event root.
- `currentValidatorSetHash` (bytes32): The current validator set hash.
- `nextValidatorSetHash` (bytes32): The next validator set hash.
- `blockHash` (bytes32): The block hash.
- `bitmap` (bytes): The bitmap.
- `signature` (bytes): The signature.

Functions

commitEpoch

Commits an epoch to the contract. Called by the Edge client.

- `id` (uint256): The ID of the epoch to be committed.
- `epoch` (Epoch): The epoch data to be committed.
- `uptime` (Uptime): The uptime data for the epoch being committed.

commitEpochWithDoubleSignerSlashing

Commits an epoch and slashes double signers. Called by the Edge client.

- `curEpochId` (uint256): The ID of the epoch to be committed.
- `blockNumber` (uint256): The block number at which the double signer occurred.
- `pbftRound` (uint256): The round number at which the double signing occurred.
- `epoch` (Epoch): The epoch data to be committed.
- `uptime` (Uptime): The uptime data for the epoch being committed.
- `inputs` (DoubleSignerSlashingInput[]): Information about double signers to be slashed along with signatures and bitmap.

getCurrentValidatorSet

Returns the addresses of active validators in the current epoch, sorted by total stake (self-stake + delegation).

getEpochByBlock

Looks up an epoch by block number in O(log n) time.

- `blockNumber` (uint256): The block number.

totalActiveStake

Calculates the total stake of active validators (self-stake + delegation).

id: validator title: Validator Interface sidebar_label: Validator description: "Learn about the access control module and how to allowlist validators." keywords: - docs - polygon - edge - consensus - interface

The `IValidator` interface represents a validator and its reward pool for delegators on the childchain. This user guide will explain the structs that make up the interface.

Structs

RewardPool

RewardPool represents a pool for reward distribution, formed by delegators to a specific validator. It tracks slashed delegations using virtual balances.

- `supply` (uint256): The amount of tokens in the pool.
- `virtualSupply` (uint256): The total supply of virtual balances in the pool.
- `magnifiedRewardPerShare` (uint256): A coefficient to aggregate rewards.
- `validator` (address): The address of the validator the pool is based on.
- `magnifiedRewardCorrections` (mapping): Adjustments to reward magnifications by address.
- `claimedRewards` (mapping): The amount claimed by address.
- `balances` (mapping): The virtual balance by address.

Validator

Validator represents a validator on the childchain.

- `blsKey` (uint256[4]): The public BLS key of the validator.
- `stake` (uint256): The amount staked by the validator.
- `commission` (uint256): The fee taken from delegators' rewards and given to the validator.
- `withdrawableRewards` (uint256): The amount that can be withdrawn.
- `active` (bool): Indicates if the validator is actively proposing/attesting.

Node

Node represents a node in the red-black validator tree.

- `parent` (address): The address of the parent of this node.
- `left` (address): The node in the tree to the left of this one.
- `right` (address): The node in the tree to the right of this one.
- `red` (bool): A boolean denoting the color of the node for balancing.
- `validator` (Validator): The validator data type.

ValidatorTree

ValidatorTree represents a red-black validator tree.

- `root` (address): The root of the tree.
- `count` (uint256): The number of nodes in the tree.
- `totalStake` (uint256): The total amount staked by nodes of the tree.
- `nodes` (mapping): A mapping from an address to a Node.
- `delegationPools` (mapping): A mapping from a validator address to a RewardPool.

---

id: supernets-allowlist-add-remove title: How to Add & Remove Accounts from an ACL sidebar_label: How to Add & Remove Accounts description: "Learn how to allowlist addresses on a Supernet." keywords: - docs - polygon - allowlist - allow - access - access control - block - blocklist

Understanding ACL and Address Roles

ACLs or Access Control Lists are a way of managing permissions in your Supernet. In this context, an ACL is essentially a list of addresses and their corresponding roles.

Roles can be one of three types:

- `NoRole` : The address has no permissions.
- `EnabledRole` : The address has some permissions.
- `AdminRole` : The address has all permissions and can change the roles of other addresses.

For more information about how ACLs work in Supernets, check out the overview guide here.

:::info Keep in mind

- **Enabling Lists**: Allowlists and blocklists are enabled or disabled exclusively during network initialization through the genesis command. Changes to the configuration cannot be made dynamically.
- **Admin Role**: To enable a list, an admin role must be set in the genesis command. The admin manages the list and can only be specified during the network's initial setup.
- **Exclusive Enablement**: It is not valid to enable both allowlists and blocklists for a given list type. If both lists are set, the allowlist takes precedence, and the blocklist is ignored.
- **System Transaction Address**: The system transaction address (0xffffFFFfFFffffffffffffffffFfFFFffffFFFfFFfE) is excluded from allowlist and blocklist validation. It is always allowed to perform actions and is not subject to list checks.
- **Impact on Validators and System Transactions**: The impact of allowlists and blocklists on validators and system transactions can vary depending on network implementation.

:::

Interacting with an ACL

Supernets' ACLs are implemented as precompiles, which are built-in contracts within the Edge client. You can interact with these ACL precompiles using libraries like ether.js and web3.js.

To interact with the ACL precompiles using an external library or client, follow these steps:

1. Obtain the ABI (Application Binary Interface) for the specific ACL precompile you want to interact with. The ABI contains the definitions of the precompile's functions, including their parameters and return types.

2. Create a contract instance for the ACL precompile using the ABI and the provider connected to the Supernet.

3. Use the contract instance to call the functions defined in the precompile's ABI. These functions allow you to manage the ACL by adding or removing accounts, modifying roles, or performing other ACL-related operations.

Example Interaction with ethers.js

Prerequisites

- Install ethers.js in your project using npm or yarn: `npm i --save ethers`.
- Ensure you have a wallet and some gas for transaction fees.
- Obtain the address and ABI of the ACL smart contract you want to interact with.

:::note

- These operations should be performed by an account with the `AdminRole`.
- Make sure to replace the placeholders with your actual values.
- The await keyword must be used in an async function.
- Be mindful of transaction fees (gas costs) when interacting with the network.

:::

Sample workflow

Create a contract instance for the ACL precompile using the ABI and the provider connected to the network. Please note that these examples are provided as guidance and assume the usage of ether.js, with Alice and Bob used as placeholders for specific agents.

```
const ethers = require('ethers');

// Connect to the provider (e.g., local node, Infura, Alchemy, etc.)
const provider = new ethers.providers.JsonRpcProvider('https://rpc-endpoint.io');

const ACLInterface = [{...}]; // Replace with the ABI of the Access List precompile

// Contract Deployer Allow List
const contractDeployerAlowListAddress = '0xContractDeployerAllowListAddress';
const contractDeployerAllowList = new ethers.Contract(contractDeployerAddress, ACLInterface, provider);

// Transactions Allow List
const transactionsAllowListAddress = '0xTransactionsAllowListAddress';
const transactionsAllowList = new ethers.Contract(transactionsAllowListAddress, ACLInterface, provider);

// Transactions Block List
const transactionsBlockListAddress = '0xTransactionsBlockListAddress';
const transactionsBlockList = new ethers.Contract(transactionsBlockListAddress, ACLInterface, provider);

// Bridge Allow List
const bridgeAllowListAddress = '0xBridgeAllowListAddress';
const bridgeAllowList = new ethers.Contract(bridgeAllowListAddress, ACLInterface, provider);

// Bridge Block List
const bridgeBlockListAddress = '0xBridgeBlockListAddress';
const bridgeBlockList = new ethers.Contract(bridgeBlockListAddress, ACLInterface, provider);
```

Interact with the ContractDeployer ACL

```
// Example interaction with the Contract Deployer ACL

const addrAlice = '0xAliceAddress'; // Replace with Alice's address
const addrBob = '0xBobAddress'; // Replace with Bob's address

// Add Alice's address to the contract deployer allowlist
const allowlistContractDeployerTx1 = contractDeployerAllowList.setEnabled(addrAlice);
allowlistContractDeployerTx1.wait();

// Remove Alice's address from the contract deployer allowlist
const allowlistContractDeployerTx2 = contractDeployerAllowList.setNone(addrAlice);
allowlistContractDeployerTx2.wait();

// Add Bob's address to the contract deployer allowlist
const allowlistContractDeployerTx3 = contractDeployerAllowList.setEnabled(addrBob);
allowlistContractDeployerTx3.wait();

// Remove Bob's address from the contract deployer allowlist
const allowlistContractDeployerTx4 = contractDeployerAllowList.setNone(addrBob);
allowlistContractDeployerTx4.wait();
```

Interact with Transactions ACL

```
// Add Bob's address to the transaction allowlist
const allowlistTxTx1 = transactionsAllowList.setEnabled(addrBob);
allowlistTxTx1.wait();

// Remove Bob's address from the transaction allowlist
const allowlistTxTx2 = transactionsAllowList.setNone(addrBob);
allowlistTxTx2.wait();

// Add Alice's address to the transaction blocklist
const blocklistTxTx1 = transactionsBlockList.setEnabled(addrAlice);
blocklistTxTx1.wait();

// Remove Alice's address from the transaction blocklist
const blocklistTxTx2 = transactionsBlockList.setNone(addrAlice);
blocklistTxTx2.wait();
```

Interact with the Bridge ACL

```
// Add Alice's address to the bridge allowlist
const allowlistBridgeTx1 = bridgeAllowList.setEnabled(addrAlice);
allowlistBridgeTx1.wait();

// Remove Alice's address from the bridge allowlist
const allowlistBridgeTx2 = bridgeAllowList.setNone(addrAlice);
allowlistBridgeTx2.wait();

// Add Alice's address to the bridge blocklist
const blocklistBridgeTx1 = bridgeBlockList.setEnabled(addrAlice);
blocklistBridgeTx1.wait();

// Remove Alice's address from the bridge blocklist
const blocklistBridgeTx2 = bridgeBlockList.setNone(addrAlice);
blocklistBridgeTx2.wait();
```

---

id: supernets-custom-bridge title: How to Use a Custom Bridge sidebar_label: How to Use a Custom Bridge description: "Learn how to disable the native bridge through allowlisting." keywords: - docs - polygon - supernets - bridge - custom - disable

Before we dive in, it is important to note that it is not possible to fully disable the native bridge. What we can do instead is significantly limit its usage through the use of allowlists.

Limiting the Use of the Native Bridge

The Supernet deployer can restrict the native bridge usage by defining an address of the bridge allow list admin via the `--bridge-allow-list-admin` flag at genesis. This action deploys allowlist predicates which verify each bridge transaction. If a sender is not permitted, the transaction will fail and revert.

As a deployer, setting your own address as the `Admin` and not including any other address in the allowlist effectively denies all other addresses from using the native bridge.

Implementing a Third-Party Bridge

When it comes to implementing a third-party bridge, it is necessary to utilize the state sync mechanism. This can be done through the `StateSender` and `L2StateSender` contracts, employing the `syncState` functions. This method permits the transmission of arbitrary data (in byte array form) from rootchain to childchain and vice versa.

Remember that receiver contracts must implement onStateReceive functions for receivers on layer 2, and onL2StateReceive for receivers on layer 1.

Flow for L1 to L2

Here are the steps to bridge from rootchain to childchain:

1. **Initiate Transaction**: The user initiates a transaction on L1 by specifying the StateSender contract, the receiver on L2, and the data to be sent.
2. **Event Emission**: An event, state sync event, is emitted and detected by validators on L2.
3. **Commitment Creation**: Validators generate a commitment from the state sync events and commit it to the StateReceiver contract on L2. This state transaction is automatically created by validators on each sprint block, given there are state sync events to commit.
4. **Event Execution**: After the commitment is submitted, the Relayer executes state sync events on the StateReceiver contract. This requires a node to be initiated with the --relayer flag.
5. **Data Reception**: During the execution step on StateReceiver, the onStateReceive function is called on the receiver contract, passing the byte array (data) from step 1.

Flow for childchain to rootchain

For bridging from childchain back to rootchain, follow these steps:

1. **Initiate Transaction**: The user sends a transaction on the childchain to the L2StateSender contract.
2. **Exit Event Emission**: An exit event is emitted andpicked up by validators on L2. The ID of this exit event can be found in the transaction receipt, which will be used in the subsequent steps.
3. **Checkpoint Creation**: Validators generate a checkpoint from these exit events and submit it to the CheckpointManager contract on L1. This occurs at the end of every epoch if there are exit events to checkpoint.
4. **Proof of Exit Generation**: Once the checkpoint containing their exit event is submitted, the user can call the bridge_generateExitProof jsonRPC endpoint on an L2 node using the exit event ID from step 2. This returns a proof of exit for the user and some additional data needed to complete the data bridge back to L1.
5. **Data Bridging**: The user then sends a transaction to the ExitHelper contract (function exit), using the data returned by the jsonRPC endpoint in the previous step. This action bridges the user's data back to L1.

---

id: supernets-genesis-validators title: How to Configure the Initial Validator Set sidebar_label: How to Configure the Initial Validator Set description: "Learn how to configure the initial validator set of a new Supernet with allowlists and staking." keywords: - docs - polygon - supernets - validators - validator set - allowlist - stake

In this section, we'll look at how to configure the initial rootchain validator set through allowlisting and by staking.

1. Allowlist validators on the rootchain

The `CustomSupernetManager` contract on the rootchain is responsible for managing the PolyBFT network.

Before validators can be registered on the `CustomSupernetManager` contract on the rootchain, they need to be allowlisted by the deployer of the `CustomSupernetManager` contract. Validators can register themselves, or a registrator can register them on their behalf. Once registered, validators can stake and start validating transactions.

This can be done using the `polygon-edge polybft whitelist-validators` command. The deployer can specify the hex-encoded private key of the `CustomSupernetManager` contract deployer or use the `--data-dir` flag if they have initialized their secrets.

▶ Flags ↓

In the following example command, we are using a placeholder private key for the `CustomSupernetManager` contract deployer.

> *If running the demo geth instance, the test account private key has been hardcoded:* `aa75e9a7d427efc732f8e4f1a5b7646adcc61fd5bae40f80d13c8419c9f43d6d`.

The `--addresses` flag is a comma-separated list of the first two validators generated earlier. The `--supernet-manager` flag specifies the actual `CustomSupernetManager` contract address that was deployed.

```
./polygon-edge polybft whitelist-validators \
  --private-key 0x0123456789abcdef0123456789abcdef0123456789abcdef0123456789abcdef \
  --addresses 0x61324166B0202DB1E7502924326262274Fa4358F,0xFE5E166BA5EA50c04fCa00b07b59966E6C2E9570 \
  --supernet-manager 0x75aA024A2292A3FD3C17d67b54B3d00435437246
```

2. Register validators on the rootchain

Each validator needs to register themselves on the `CustomSupernetManager` contract. This is done using the `polygon-edge polybft register-validator` command. **Note that this command is for testing purposes only.**

▶ Flags ↓

```
./polygon-edge polybft register-validator --data-dir ./test-chain-1 \
--supernet-manager --supernet-manager 0x75aA024A2292A3FD3C17d67b54B3d00435437246
```

3. Initial staking on the rootchain

Each validator needs to perform initial staking on the rootchain `StakeManager` contract. This is done using the `polygon-edge polybft stake` command. **Note that this command is for testing purposes only.**

▶ Flags ↓

In the following example command, we use the validator key and the rootchain `StakeManager` contract address that was generated earlier. We also set the staking amount to `1000000000000000000` which is equivalent to 1 token. The `--native-root-token` flag is used to specify the address of the native token of the rootchain.

:::info Staking requirement: wrapping a non-ERC-20 token

Supernets allow for the customization of the gas token and mandate the use of ERC-20 tokens for staking instead of the rootchain's native token.

When performing rootchain staking on the Polygon PoS Mainnet, WMATIC (wrapped MATIC) is the required token. This is due to the ERC-20 standard requirement for staking, which MATIC doesn't meet.

WMATIC is deployed at `0x0d500b1d8e8ef31e21c99d1db9a6444d3adf1270`.

This principle also applies to the Mumbai Testnet, where wrapped test MATIC must be used instead of test MATIC.

If you currently hold MATIC tokens, you can convert them to WMATIC through various methods. One common method is to use a decentralized exchange (DEX) like Uniswap, where you can swap your MATIC tokens for WMATIC. **Always ensure you're using a reputable platform for this conversion and double-check that the contract address for WMATIC is correct.**

▶ Obtaining native root token address

:::

```
./polygon-edge polybft stake \
--data-dir ./test-chain-1 \
--chain-id 100 \
--amount 1000000000000000000 \
--stake-manager 0x6ceCFe1Db48Ab97fA89b06Df6Bd0bBdE6E64e6F7 \
--native-root-token 0x559Dd13d8A3CAb3Ff127c408f2A08A8a2AEfC56c
```

4. Finalize validator set on the rootchain

After all validators from the genesis block have performed initial staking on the rootchain, the final step required before starting the chain is to finalize the genesis validator set on the `SupernetManager` contract on the rootchain. This can be done using the `polygon-edge polybft supernet` command.

The deployer of the `SupernetManager` contract can specify their hex-encoded private key or use the `--data-dir` flag if they have initialized their secrets. If the `--enable-staking` flag is provided, validators will be able to continue staking on the rootchain. If not, genesis validators will not be able to update their stake or unstake, nor will newly registered validators after genesis be able to stake tokens on the rootchain. The enabling of staking can be done through this command or later after the Supernet starts.

In the following example command, we use a placeholder hex-encoded private key of the `SupernetManager` contract deployer. The addresses of the `SupernetManager` and `StakeManager` contracts are the addresses that were generated earlier. We also use the `--finalize-genesis` and `--enable-staking` flags to enable staking and finalize the genesis state.

```
./polygon-edge polybft supernet --private-key 0x0123456789abcdef0123456789abcdef0123456789abcdef0123456789abcdef \
--genesis /path/to/genesis/file \
--supernet-manager 0x75aA024A2292A3FD3C17d67b54B3d00435437246 \
--stake-manager 0x811068e4106f7A70D443684FF4927eC3940439Ec \
--finalize-genesis --enable-staking
```

5. Next Steps

With all the necessary configurations in place for the Supernet, we are ready to proceed with starting the chain.

Navigate to the Start Your Supernet deployment guide, which will provide you with instructions on how to initiate and launch the chain.

id: supernets-how-to-generate-genesis title: How to Configure a New Childchain sidebar_label: How to Configure a New Childchain description: "Learn how to deploy a local test Supernet." keywords: - docs - polygon - edge - supernets - network - modular

import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem';

In this section, we'll walk through how to configure the initial childchain state by generating a new genesis file.

1. Overview

The genesis file is a critical component in setting up a blockchain network, containing the initial validator set, genesis block, and other essential parameters that define the network's behavior. The initial validator set is responsible for bootstrapping the consensus mechanism, allowing the blockchain to function and reach consensus on new blocks.

Supernets allow for customizable parameters such as the block gas limit, epoch size, and block rewards, which enable network operators to tailor the network to specific requirements. Additionally, Supernets support allowlists and blocklists for transactions and validators, providing an extra layer of control and security to the network. These lists can be used to restrict or permit specific addresses, ensuring only authorized parties can participate in the network or execute transactions.

To create the chain configuration, we use the `polygon-edge genesis` command, which generates the genesis file.

▶ Common Flags ↓
▶ Access Control List Flags ↓
:::note ACL gas cost considerations

While the use of alternative ACL-enabled contracts, such as bridge ACLs, offers finer control over cross-chain interactions, these contracts also result in increased gas consumption for transactions. As you weigh the benefits of enhanced security, keep in mind that security measures can often come with higher costs.

:::

It's crucial to consider these parameters carefully, as they have a significant impact on the network's performance, security, and scalability. Informed decisions are vital to building a robust and efficient blockchain network.

:::caution Must enable ACLs before launching the network

Keep in mind that allowlists must be enabled prior to launching the network. After the network is started, configuration of the allowlist will no longer be possible. The access control process will be initiated by the initial allowlisted addresses.

:::

2. Deployment Considerations

i. Difference between `chain-id` and `supernet-id`

Supernets differentiates `chain-id` and `supernet-id` within the genesis file.

- The `chain-id` is a unique identifier for a childchain.
- The `supernet-id` is a unique identifier that is automatically populated during the Supernet's registration process with the `StakeManager` contract. There can be multiple Supernets registered within the same `StakeManager` - the `supernet-id` is used to distinguish them.

Users can assign a custom `chain-id` to their Supernet via the genesis command. However, `supernet-id` doesn't function as a flag within this command. Instead, its value is derived from the `rootchain deploy` command during the Supernet's registration process with the `StakeManager` contract.

ii. Create a Native Token and Premine

Flag	Description
`--premine`	Specify premined accounts and their balances. Use the flag multiple times for multiple accounts.
`--native-token-config`	Configure the native token, including its name, symbol, decimal count, and whether it is mintable.

> *There is also an `--owner` flag only applicable for mintable native tokens and designates the account that has the permission to mint and burn tokens, in addition to the predicate.*

For example, the following command creates a native token named `MyToken` with the symbol `MTK`, `18` decimal places, and a total supply of `1,000,000` tokens. It also premines `1,000` tokens to the account at address `0x61324166B0202DB1E7502924326262274Fa4358F`.

▶ Example ↓

> *Note that you can omit the `--native-token-config` flag if you don't want to create a native token. For non-mintable native tokens, premining is limited to the `0x0` address - other accounts must bridge over assets from the rootchain.*

iii. Enable EIP1559

Flag	Description
`--burn-contract`	Specify the burn contract address to enable the London hard fork and set where fees will be sent.
`--genesis-base-fee`	Set the initial base fee (in GWEI) for the genesis block.

For example, the following command enables EIP1559 by specifying the burn contract address and setting the genesis base fee to 2 GWEI:

> *The burn contract address must be specified in the format `<block>:<address>`. The genesis base fee value is in GWEI.*

▶ Example ↓

3. Specify Validator Set & Generate Genesis

There are two ways to specify the initial validator set:

- **Single-Host Validator Setup**: All validator information is present in the local storage of a single host. In this case, you can provide the directory using the `--validators-path` flag and the validator folder prefix names using the `--validators-prefix` flag. To enable reward distribution, you must define a

reward wallet address and its balance using the `--reward-wallet` flag. The wallet address is used to distribute reward tokens from that address to validators that signed blocks in that epoch.

- **Multi-Host Validator Setup**: Validator information is scattered across multiple hosts. In this case, you can supply the necessary information using the `--validators` flag.

<Tabs defaultValue="option-1" values={[{ label: 'Single-Host Validator Setup', value: 'option-1', }, { label: 'Multi-Host Validator Setup', value: 'option-2', },] }>

In the following example command, we will set the block gas limit to `10,000,000` and the epoch size to `10` using the `--block-gas-limit` and `--epoch-size` flags, respectively.

We will also specify the consensus mechanism as PolyBFT using the `--consensus` flag.

To enable reward distribution, we will define a reward wallet address and its balance using the `--reward-wallet` flag (for this example, we use the first validator as the reward wallet).

In addition, we can provide the directory where the validator information is stored using the `--validators-path` flag and the prefix for the validator folder names using the `--validators-prefix` flag.

We also add the `--transactions-allow-list-admin` flag to specify the admin addresses for the transactions allowlist, and the `--transactions-allow-list-enabled` flag to specify the addresses that will be enabled by default in the transactions allowlist, using the first and second validator nodes.

You can customize the initial chain state using the flags listed above.

```
./polygon-edge genesis --block-gas-limit 10000000 --epoch-size 10 \
    --validators-path ./ --validators-prefix test-chain- \
    --consensus polybft \
    --reward-wallet 0x61324166B0202DB1E7502924326262274Fa4358F:1000000 \
    --transactions-allow-list-admin 0x61324166B0202DB1E7502924326262274Fa4358F,0xFE5E166BA5EA50c04fCa00b07b59966E6C2E9570 \
    --transactions-allow-list-enabled 0x61324166B0202DB1E7502924326262274Fa4358F,0xFE5E166BA5EA50c04fCa00b07b59966E6C2E9570
```

By customizing these flags, we can tailor the network to meet our specific requirements. Remember to consider the impact of these parameters carefully, as they can significantly affect the network's performance, security, and scalability.

▶ Genesis output example ↓

When setting up the initial chain state for a multiple-host validator setup, you'll need to specify the validator information using the `--validators` flag in the following format: `<multi_address>:<public_ECDSA_address>:<public_BLS_key>:<BLS_signature>`.

This follows the [libp2p multiaddr format](): `<ip4><ip_address><transport><port><p2p><node_id>`.

In this example, we will designate the first and second nodes as bootnodes for all other nodes. Nodes connecting to node 1 or 2 will receive information on how to connect to each other through the mutually contacted bootnode. For the ports, we will use `30301` for the first node and `30302` for the second node. Since we are running on localhost, it is safe to assume that the `ip_address` is `127.0.0.1`. If you are not running on your localhost, please update accordingly.

After assembly, the multiaddr connection string for node 1 and 2 will look something like this:

- Node 1: `/ip4/127.0.0.1/tcp/30301/p2p/16Uiu2HAmMYyzK7c649Tnn6XdqFLP7fpPB2QWdck1Ee9vj5a7Nhg8`
- Node 2: `/ip4/127.0.0.1/tcp/30302/p2p/16Uiu2HAmLXVapjR2Yx3B1taCmHnckQ1ph2xrawBjW2kvSErps9CX`

In the following example command, we will set the block gas limit to `10,000,000` and the epoch size to `10` using the `--block-gas-limit` and `--epoch-size` flags, respectively.

We will also specify the consensus mechanism as PolyBFT using the `--consensus` flag.

To enable reward distribution, we will define a reward wallet address and its balance using the `--reward-wallet` flag (for this example, we use the first validator as the reward wallet).

In addition, we can provide the directory where the validator information is stored using the `--validators-path` flag and the prefix for the validator folder names using the `--validators-prefix` flag.

We also add the `--transactions-allow-list-admin` flag to specify the admin addresses for the transactions allowlist, and the `--transactions-allow-list-enabled` flag to specify the addresses that will be enabled by default in the transactions allowlist, using the first and second validator nodes.

You can customize the initial chain state using the flags listed above.

```
./polygon-edge genesis --block-gas-limit 10000000 --epoch-size 10 \
    --validators
"/ip4/127.0.0.1/tcp/30301/p2p/16Uiu2HAmMYyzK7c649Tnn6XdqFLP7fpPB2QWdck1Ee9vj5a7Nhg8:0x61324166B0202DB1E7502924326262274Fa4358F:06d8
\
    --validators
"/ip4/127.0.0.1/tcp/30302/p2p/16Uiu2HAmLXVapjR2Yx3B1taCmHnckQ1ph2xrawBjW2kvSErps9CX:0xFE5E166BA5EA50c04fCa00b07b59966E6C2E9570:0603
\
    --consensus polybft \
    --reward-wallet 0x61324166B0202DB1E7502924326262274Fa4358F:1000000 \
    --transactions-allow-list-admin 0x61324166B0202DB1E7502924326262274Fa4358F,0xFE5E166BA5EA50c04fCa00b07b59966E6C2E9570 \
    --transactions-allow-list-enabled 0x61324166B0202DB1E7502924326262274Fa4358F,0xFE5E166BA5EA50c04fCa00b07b59966E6C2E9570
```

By customizing these flags, we can tailor the network to meet our specific requirements. Remember to consider the impact of these parameters carefully, as they can significantly affect the network's performance, security, and scalability.

▶ Genesis output example

4. Next Steps

With a **genesis.json** file containing the initial chain state, validator nodes, and chain admins for your new childchain instance, you are ready to proceed.

To configure the associated rootchain of the Supernet and deploy the essential rootchain core contracts, navigate to the [rootchain deployment guide](#).

id: supernets-how-to-upgrade title: How to Upgrade Your Supernet Using a Hardfork sidebar_label: Upgrades description: "Learn how to upgrade your Supernet using a hardfork." keywords: - docs - polygon - edge - supernets - uograde - hardfork

Drawing from a real-world example, this guide delves into the process of introducing a new feature via a hardfork in a Supernet, utilizing a fork manager to manage the activation and behavior of the fork.

1. Adding Fields

Add the `ExtraData` field to Extra. Before block 100, the field will be ignored. After block 100, it will be used. Your Extra struct should look like this:

```
type Extra struct {
    Validators  *validator.ValidatorSetDelta
    Parent      *Signature
    Committed   *Signature
    Checkpoint  *CheckpointData
    ExtraData   []byte // NewField
    BlockNumber uint64 // field used by forking manager
}
```

The `ExtraData` field is a byte slice (`[]byte`), which can hold any kind of data.

2. Implement Extra.Marshal() Behavior

The marshaling process will need to account for the ExtraData field after block 100.

```
type ExtraHandlerNewField struct {
}

// MarshalRLPWith defines the marshal function implementation for Extra
func (e *ExtraHandlerNewField) MarshalRLPWith(extra *Extra, ar *fastrlp.Arena) *fastrlp.Value {
    vv := extraMarshalBaseFields(extra, ar, ar.NewArray())
    vv.Set(ar.NewBytes(extra.ExtraData))
    return vv
}
```

This code creates a new type `ExtraHandlerNewField` which has a method `MarshalRLPWith` . This method handles the marshaling of the `Extra` struct and specifically takes into account the `ExtraData` field.

3. Register the Fork in chain/params.go

`chain/params.go` is where the details about the fork are specified, such as its name, when it is activated, and how to determine its activity at a given block.

3.1 Define the Fork's Name

We first define a constant string to represent the name of the fork.

```
const (
...
    NewFieldForkName     = "newFieldFork"
)
```

3.2 Add the Fork as a Supported Fork

We then include the fork in the `ForksInTime` struct to mark it as a supported fork.

```
type ForksInTime struct {
...
    NewFieldFork bool
}
```

3.3 Include the Fork into the genesis.json

Next, we include the fork in the `genesis.json` .

```go
var AllForksEnabled = &Forks{
...
    NewFieldForkName:    NewFork(0),
}
```

3.4 Add Support to Read if the Fork is Active at the Current Block

Finally, we add the ability to read if the fork is active at a specific block.

```go
func (f *Forks) At(block uint64) ForksInTime {
    return ForksInTime{
...
        NewFieldFork:    f.IsActive(NewFieldForkName, block),
    }
}
```

4. Register Fork Handlers

Fork handlers are the functions that are run when a fork becomes active. Here, we register a handler for our new fork, as well as a default handler for blocks before the fork.

```go
func (f *Forks) At(block uint64) ForksInTime {
    return ForksInTime{
...
        NewFieldFork:    f.IsActive(NewFieldForkName, block),
    }
}
```

5. Fork Management

In the `chain/params.go` file, define the parameters related to the fork. They should always be pointer values. For example:

```go
type ForkParams struct {
    MaxValidatorSetSize *uint64 `json:"maxValidatorSetSize,omitempty"`
    ...
}
```

These parameters represent new rules or behaviors to be introduced.

6. Configuring the Fork in genesis.json

In the genesis.json file, specify all available forks and their parameters. For instance:

```json
"forks": {
    "EIP150": {
        "block": 0
    },
    ...
    "myFork": {
        "block": 10,
        "params": {
            "maxValidatorSetSize": 20
        }
    },
}
```

Here, a new fork named `myFork` is added with specific parameters and values. For this fork, `maxValidatorSetSize` will be 20.

7. Accessing and Using Fork Parameters in Code

You can access these parameters in the code using the fork manager. For example:

```go
maxValidatorSetSize := 100 // default value
ps := forkmanager.GetInstance().GetParam(blockNumber)
if ps != nil {
    maxValidatorSetSize = ps.MaxValidatorSetSize
}
```

For `blockNumber` < 10, `maxValidatorSetSize` will be set to 100. For `blockNumber` >= 10, the `maxValidatorSetSize` will be 20, the value specified for the `myFork` fork.

8. Setting Default Parameter Values

If you don't want to use if statements in the code and always want to have default values for the parameters, change the consensus factory method defined in `server/builtin.go` :

```
func ForkManagerInitialParamsFactory(config *chain.Chain) (*chain.ForkParams, error) {
    pbftConfig, err := GetPolyBFTConfig(config)
    if err != nil {
        return nil, err
    }

    return &chain.ForkParams{
        MaxValidatorSetSize: &pbftConfig.MaxValidatorSetSize,
        ...
    }, nil
}
```

Each fork doesn't need to specify all parameters, only the ones that have changed between forks. Other parameters take the value from the previous fork.

9. Network Upgrade

The introduction of a hard fork necessitates an upgrade process across the network. Nodes need to update their software to incorporate the new fork code, and the new `ExtraData` field becomes active upon reaching the designated block height. Here's an example of the step-by-step process that follows the example used above:

1. **Generate New Genesis File**: Create a new genesis.json using the updated Edge implementation that includes the fork.
2. **Edit Activation Block**: Set the fork's activation block to 100 in the genesis.json file, as in "MyFirstForkName": 100.
3. **Distribute Genesis File**: Share the updated genesis.json file with all network nodes.

   For a step-by-step guide on generating a new genesis file, please consult the chain configuration deployment guide here.
4. **Stop Client**: Pause the Edge client on all network nodes before block 100.
5. **Replace Files**: Replace the existing genesis.json and Edge files with the updated versions.
6. **Restart Edge**: Restart the Edge client to apply the hard fork updates.

Remember, communication with network participants about the specific process and timeline is crucial for a successful upgrade.

id: supernets-deploy-index title: Get Started sidebar_label: Get Started description: "Learn how to deploy a local test Supernet." keywords: - docs - polygon - edge - supernets - network - modular

import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem';

This serves as an index for the Supernet deployment guides. The guides explain the end-to-end process for setting up and deploying a local Supernet.

Prerequisites

Before diving into any of the tutorials, make sure your environment meets the necessary prerequisites. They can be found **here**.

▶ Before starting ↓

What you'll learn

In this tutorial, you will learn how to set up and initialize a Supernet with multiple nodes. You will learn the complete end-to-end genesis workflow of building a Supernet, including:

- Generating private keys for PolyBFT nodes.
- Deploying and initializing rootchain contracts.
- Configuring the rootchain by allowlisting and registering validators, performing initial staking, and finalizing the validator set.
- Generating the genesis file and chain configuration.
- Funding validators on the rootchain.
- Running a (Supernet) cluster consisting of multiple PolyBFT nodes.

By the end of this tutorial, you will have a fully functional Supernet test network that can be used to process transactions with high throughput and low latency.

What you'll do

The deployment guides will cover the steps outlined below. If you are deploying a Supernet for the first time, please navigate each section in order.

1. Spawn a New Childchain
2. Configure Your New Childchain
3. Configure the Rootchain
4. Configure the Initial Validator Set
5. Start Your New Supernet

▶ Fast-track guide ↓

Cloud Deployments

Platform	Guide
Amazon Web Services	To set up a devnet on AWS, you can refer to the AWS deployment guide available here. The guide provides comprehensive instructions on how to use Terraform to set up a Virtual Private Cloud (VPC), subnets, security groups, and EC2 instances, followed by instructions on configuring nodes using Ansible.
Microsoft Azure	To set up a devnet on Azure, you can refer to the Azure deployment guide available here. This repository offers an Azure template that can be deployed through the Azure and Bicep CLI, or directly through the "Deploy to Azure" button. Additionally, the deployment can be viewed via the "Visualize" button available in the repository.
Google Cloud Platform	Coming soon.

---

id: supernets-spawn-test-chain title: How to Generate New Account Secrets sidebar_label: How to Spawn a New Childchain description: "Learn how to deploy a local test Supernet." keywords: - docs - polygon - edge - supernets - network - modular

In this section, we'll prepare initiate a new chain with PolyBFT consensus and prepare the initial Supernet nodes.

1. Generate Keys

To initialize PolyBFT consensus, we need to generate the necessary secrets for each node.

The `polygon-edge polybft-secrets` command is used to generate account secrets for validators. The command initializes private keys for the consensus client (validators + networking) to a Secrets Manager config file.

Keep in mind that this command is intended for testing purposes only and should not be used in a production environment.

▶ Flags ↓

```
./polygon-edge polybft-secrets --insecure --data-dir test-chain- --num 4
```

▶ Output example ↓

Understand the generated secrets

The generated secrets include the following information for each validator node:

- **ECDSA Private and Public Keys**: These keys are used to sign and verify transactions on the blockchain.
- **BLS Private and Public Keys**: These keys are used in the Byzantine fault-tolerant (BFT) consensus protocol to aggregate and verify signatures efficiently.
- **P2P Networking Node ID**: This is a unique identifier for each validator node in the network, allowing them to establish and maintain connections with other nodes.

> *The secrets output can be retrieved again if needed by running the following command:* `./polygon-edge polybft-secrets --data-dir test-chain-X/ --insecure`

2. Next Steps

As the next step, navigate to the Configure a New Childchain deployment guide. This will enable you to generate a new genesis file and define the initial validator set, facilitating the setup and configuration of the childchain's initial state.

id: supernets-how-to-configure-rootchain title: How to Configure the Rootchain sidebar_label: How to Configure the Rootchain description: "Learn how to configure the rootchain that is associated with your Supernet." keywords: - docs - polygon - supernets - rootchain - configuration - PoS - Geth

import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem';

In this section, we'll configure the associated rootchain of the Supernet and deploy the necessary rootchain core contracts.

After generating the initial chain state for your Supernet, the next step is to connect and initialize the rootchain contracts. This can be done using either a demo Geth instance or any EVM-compatible rootchain. The demo Geth instance is a local instance of a Geth node running in development mode, which simulates the Ethereum network and is **only intended for testing purposes**.

1. Deploy and initialize rootchain contracts

<Tabs defaultValue="geth" values={[{ label: 'Demo Geth instance', value: 'geth', }, { label: 'EVM-compatible rootchain', value: 'rootchain', },] }>

i. Start the geth node

The `polygon-edge` rootchain server command starts an ethereum/client-go container, which runs a new Geth node. To do this, open a new terminal session and run:

```
./polygon-edge rootchain server
```

▶ Output example ↓

This will start the rootchain server on the default JSON-RPC port of `8545`.

ii. Deploy StakeManager contract

:::caution

If you have already deployed the StakeManager, you may skip the step and move onto Deploy rootchain contracts.

:::

If the `StakeManager` hasn't been deployed to the rootchain, you need to carry out this step. This command also contains a test flag. This flag is strictly for testing purposes, and its usage results in deploying a mock ERC-20 token that will serve for staking.

```
./polygon-edge polybft stake-manager-deploy \
--private-key <hex_encoded_rootchain_account_private_key> \
--genesis ./genesis.json \
--jsonrpc http://127.0.0.1:8545 \
--stake-token 0xaddressOfStakeToken \
--test
```

▶ Flags ↓

2. Deployment Considerations

To deploy the rootchain contracts, we use the `polygon-edge rootchain deploy` command. Before deployment, consider the following actions:

i. Funding required for nodes

Before initializing the contracts on the rootchain, we need to make sure that the nodes are funded with sufficient funds to cover the gas cost of deploying the contracts. Otherwise, the initialization process may fail due to a lack of funds.

Note that the demo server already funds the default test account. If you're not omitting `--deployer-key`, ensure that you pass the correct key; otherwise, you may encounter an error due to insufficient funds, such as the following:

```
failed to deploy rootchain contracts: {"code":-32000,"message":"INTERNAL_ERROR: insufficient funds"}
```

You can also create a rootchain wallet and fund the nodes by using `polygon-cli`. Follow the steps outlined here.

ii. Using an existing ERC-20 as the native gas token

If you already have an ERC-20 token deployed on the rootchain that you want to use as the gas token, you can specify its address using the `--erc20-token` flag when deploying the rootchain contracts. For example:

▶ Example ↓

Replace `<ERC20_TOKEN_ADDRESS>` with the address of your existing ERC-20 token on the rootchain.

To deposit a desired amount and mint it on the childchain, please refer to the guidelines outlined in the deposit guide here.

3. Deploy rootchain contracts

Using the `--deployer-key` flag, you will need to replace `<hex_encoded_deployer_private_key>` with the hex-encoded private key of the deployer account that will be used to deploy the smart contracts. If you omit the `--deployer-key` option, the default account in your local client will be used.

> *The user is responsible for providing the deployer key, which should correspond to an address with sufficient funds for deployment. It is recommended to ensure the account is pre-funded before initiating the deployment process.*

You also need to specify the path to the genesis file using the `--genesis` option, and the endpoint for the JSON-RPC endpoint for the rootchain using the `--json-rpc` option.

To run the deployment in test mode and use the test account provided by the Geth dev instance as the depositor, add the `--test` flag. In this case, you may omit the `--deployer-key` flag, and the default test account will be used as the depositor.

▶ Flags ↓

```
./polygon-edge rootchain deploy \
  --genesis ./genesis.json \
  --json-rpc http://127.0.0.1:8545 \
  --test
```

▶ Core contract deployment output example

i. Deploy StakeManager contract

:::caution

If you have already deployed the StakeManager, you may skip the step and move onto Deploy rootchain contracts.

:::

This command includes a test flag, which is intended solely for testing scenarios. When this flag is used, a mock ERC-20 token is deployed for staking. However, in non-testing environments, remember to specify the `stake-token` flag with the address of the token that's already deployed on the rootchain and will be used for staking.

```
./polygon-edge polybft stake-manager-deploy \
--deployer-key <hex_encoded_rootchain_account_private_key> \
--genesis ./genesis.json \
--jsonrpc http://127.0.0.1:8545 \
--stake-token 0xaddressOfStakeToken \
--test
```

▶ Flags ↓

2. Deployment Considerations

To deploy the rootchain contracts, we use the `polygon-edge rootchain deploy` command. Before deployment, consider the following actions:

i. Funding required for nodes

Before initializing the contracts on the rootchain, we need to make sure that the nodes are funded with sufficient funds to cover the gas cost of deploying the contracts. Otherwise, the initialization process may fail due to a lack of funds.

Note that the demo server already funds the default test account. If you're not omitting `--deployer-key`, ensure that you pass the correct key; otherwise, you may encounter an error due to insufficient funds, such as the following:

```
failed to deploy rootchain contracts: {"code":-32000,"message":"INTERNAL_ERROR: insufficient funds"}
```

You can also create a rootchain wallet and fund the nodes by using `polygon-cli`. Follow the steps outlined here.

ii. Using an existing ERC-20 as the native gas token

If you already have an ERC-20 token deployed on the rootchain that you want to use as the gas token, you can specify its address using the `--erc20-token` flag when deploying the rootchain contracts. For example:

▶ Example ↓

Replace `<ERC20_TOKEN_ADDRESS>` with the address of your existing ERC-20 token on the rootchain.

To deposit a desired amount and mint it on the childchain, please refer to the guidelines outlined in the deposit guide here.

3. Deploy rootchain contracts

Using the `--deployer-key` flag, you will need to replace `<hex_encoded_deployer_private_key>` with the hex-encoded private key of the deployer account that will be used to deploy the smart contracts. If you omit the `--deployer-key` option, the default account in your local client will be used.

You also need to specify the path to the genesis file using the `--genesis` option, and the endpoint for the JSON-RPC endpoint for the rootchain using the `--json-rpc` option.

▶ Flags ↓

```
./polygon-edge rootchain deploy \
  --deployer-key 0x0123456789abcdef0123456789abcdef0123456789abcdef0123456789abcdef \
  --genesis ./genesis.json \
  --json-rpc http://127.0.0.1:8545 \
```

▶ Core contract deployment output example ↓

4. Funding validators on the rootchain

Before deploying validator nodes on the Supernet, we need to ensure that the validators have sufficient funds on the rootchain network. It's crucial to have enough funds in the validator account, as they need to cover the gas fees associated with their transactions on the rootchain.

To fund the validators' accounts on the rootchain, we use the `polygon-edge rootchain fund` command. When executed with the appropriate flags, it will:

1. Retrieve the validator account from the secrets manager.
2. Send a transaction to fund the validator account on the rootchain with a value of 10,000 $token (the network's native token).
3. Repeat steps 1 and 2 for each validator.

It's important to note that this command is also for testing purposes only.

In a production environment, you would need to ensure that the validators have sufficient funds on the rootchain network to cover the gas fees associated with their transactions.

Here's an example of how to fund a validator account (replace address value with your own):

```
./polygon-edge rootchain fund --addresses 0x77C1eedFf656477462ce16084fE5Dc7F8a2507B9 --amounts 1000000000000000000
```

▶ Funding output example ↓

5. Next Steps

Now that we have successfully configured the initial setup of the Supernet, including both the childchain and rootchain, we can proceed to the next crucial step: rootchain staking and allowlisting through the configuration of the initial validator set.

To accomplish this, navigate to the [Configure the Initial Validator Set]() deployment guide.

id: supernets-how-to-stake title: How to Stake on the Rootchain sidebar_label: How to Stake description: "Learn how to stake on the rootchain." keywords: - docs - polygon - supernets - stake - staking - MATIC - WMATIC - rootchain

In this section, we'll walkthrough how to stake WMATIC on the associated rootchain.

i. Initial staking on the rootchain

Each validator needs to perform initial staking on the rootchain `StakeManager` contract. This is done using the `polygon-edge polybft stake` command. **Note that this command is for testing purposes only.**

▶ Flags ↓

In the following example command, we use the validator key and the rootchain `StakeManager` contract address that was generated earlier. We also set the staking amount to `1000000000000000000` which is equivalent to 1 token. The `--native-root-token` flag is used to specify the address of the native token of the rootchain.

:::info Staking requirement: wrapping a non-ERC-20 token

Supernets allow for the customization of the gas token and mandate the use of ERC-20 tokens for staking instead of the rootchain's native token.

When performing rootchain staking on the Polygon PoS Mainnet, WMATIC (wrapped MATIC) is the required token. This is due to the ERC-20 standard requirement for staking, which MATIC doesn't meet.

WMATIC is deployed at `0x0d500b1d8e8ef31e21c99d1db9a6444d3adf1270`.

This principle also applies to the Mumbai Testnet, where wrapped test MATIC must be used instead of test MATIC.

If you currently hold MATIC tokens, you can convert them to WMATIC through various methods. One common method is to use a decentralized exchange (DEX) like Uniswap, where you can swap your MATIC tokens for WMATIC. **Always ensure you're using a reputable platform for this conversion and double-check that the contract address for WMATIC is correct to ensure the security of your tokens.**

▶ Obtaining native root token address

:::

```
./polygon-edge polybft stake \
--data-dir ./test-chain-1 \
--chain-id 100 \
--amount 1000000000000000000 \
--stake-manager 0x6ceCFe1Db48Ab97fA89b06Df6Bd0bBdE6E64e6F7 \
--native-root-token 0x559Dd13d8A3CAb3Ff127c408f2A08A8a2AEfC56c
```

ii. Finalize validator set on the rootchain

After all validators from the genesis block have performed initial staking on the rootchain, the final step required before starting the chain is to finalize the genesis validator set on the `SupernetManager` contract on the rootchain. This can be done using the `polygon-edge polybft supernet` command.

The deployer of the `SupernetManager` contract can specify their hex-encoded private key or use the `--data-dir` flag if they have initialized their secrets. If the `--enable-staking` flag is provided, validators will be able to continue staking on the rootchain. If not, genesis validators will not be able to update their stake or unstake, nor will newly registered validators after genesis be able to stake tokens on the rootchain. The enabling of staking can be done through this command or later after the Supernet starts.

In the following example command, we use a placeholder hex-encoded private key of the `SupernetManager` contract deployer. The addresses of the `SupernetManager` and `StakeManager` contracts are the addresses that were generated earlier. We also use the `--finalize-genesis` and `--enable-staking` flags to enable staking and finalize the genesis state.

```
./polygon-edge polybft supernet --private-key 0x0123456789abcdef0123456789abcdef0123456789abcdef0123456789abcdef \
--genesis /path/to/genesis/file \
--supernet-manager 0x75aA024A2292A3FD3C17d67b54B3d00435437246 \
--stake-manager 0x811068e4106f7A70D443684FF4927eC3940439Ec \
--finalize-genesis --enable-staking
```

---

id: supernets-unstake title: How to Unstake on the Rootchain sidebar_label: How to Unstake description: "A guide on the how to unstake validator stake." keywords: - docs - polygon - supernets - stake - unstake - withdraw

In this section, we'll go over how to unstake a validator's staked tokens and withdraw them to an external address.

Unstake

:::info Check validator information

First, check your validator information by running the `polygon-edge polybft validator-info` command.

▶ Flags ↓

```
./polygon-edge polybft validator-info --data-dir ./test-chain-1
```

This will show you information about your validator account, including the staked amount.

:::

To unstake, use the `polygon-edge polybft unstake` command.

▶ Flags ↓

```
./polygon-edge polybft unstake \
  --account-dir <DATA_DIR> \
  --jsonrpc <JSONRPC_ADDR> \
  --amount <AMOUNT>
```

Withdraw

Once you have successfully unstaked, you will need to withdraw your unstaked tokens from the Supernet to the rootchain. To do so, use the `polygon-edge polybft withdraw-child` command.

▶ Flags ↓

```
./polygon-edge polybft withdraw-child \
    --account-dir <DATA_DIR> \
    --jsonrpc <JSONRPC_ADDR>
```

This command will bridge the unstaked amount to the rootchain (`StakeManager`), where the given amount of tokens will become released on the given contract.

Wait for checkpoint

Next, you will need to wait for the exit event (bridge event) to be included in a checkpoint. You can confirm that the checkpoint has been successfully processed by checking the processed checkpoints on a blockchain explorer.

Send an exit transaction

Once the exit event has been included in a checkpoint, you can send an exit transaction to execute the transaction on the rootchain. To do so, use the `polygon-edge bridge exit` command.

▶ Flags ↓

```
./polygon-edge bridge exit \
    --sender-key <hex_encoded_txn_sender_private_key> \
    --exit-helper <exit_helper_address> \
    --exit-id <exit_event_id> \
    --root-json-rpc <root_chain_json_rpc_endpoint> \
    --child-json-rpc <child_chain_json_rpc_endpoint>
```

This will trigger the withdrawal of the specified amount of tokens from the rootchain.

Withdraw from root

Finally, you can withdraw your tokens from the rootchain to your address by using the `polygon-edge polybft withdraw-root` command.

▶ Flags ↓

```
./polygon-edge polybft withdraw-root \
  --account-dir <DATA_DIR> \
  --to <RECIPIENT_ADDRESS> \
  --amount <AMOUNT> \
  --stake-manager <STAKE_MANAGER_ADDRESS> \
  --jsonrpc <BRIDGE_JSONRPC_ADDR>
```

This will transfer your withdrawn tokens back to the specified withdrawal address.

id: supernets-how-to-start title: How to Start Your Supernet sidebar_label: How to Start Your Supernet description: "Learn how to start your new Supernet after configuring the genesis workflow." keywords: - docs - polygon - edge - supernets - network - modular

Now that we have set up the rootchain and created the chain configuration, we can run the Supernet cluster of Edge clients. In this example, we'll run a cluster of four Edge clients.

To run a Supernet cluster, we use the `polygon-edge server` command with the following options:

▶ Flags ↓

```
./polygon-edge server --data-dir ./test-chain-1 --chain genesis.json --grpc-address :5001 --libp2p :30301 --jsonrpc :10001 --
seal --log-level DEBUG

./polygon-edge server --data-dir ./test-chain-2 --chain genesis.json --grpc-address :5002 --libp2p :30302 --jsonrpc :10002 --
seal --log-level DEBUG

./polygon-edge server --data-dir ./test-chain-3 --chain genesis.json --grpc-address :5003 --libp2p :30303 --jsonrpc :10003 --
seal --log-level DEBUG

./polygon-edge server --data-dir ./test-chain-4 --chain genesis.json --grpc-address :5004 --libp2p :30304 --jsonrpc :10004 --
seal --log-level DEBUG
```

▶ Dialing output example

--- id: supernets-cross-chain-deposit title: How To Perform Cross-Chain Asset Transfers From Rootchain to Childchain sidebar_label: How to Deposit (Rootchain → Childchain) description: "A guide on the available bridge transactions." keywords: - docs - polygon - edge - supernets - bridge - crosschain - transactions - how to - how-to ---

import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem';

Prerequisites

You'll need to have a a successful bridge deployment to make any cross-chain transactions. If you haven't done so already, check out the local deployment guide here.

:::caution Key management and secure values When passing values to run transactions, it is important to keep sensitive values like private keys and API keys secure.

The sample commands provided in this guide use sample private keys for demonstration purposes only, in order to show the format and expected value of the parameter. It is important to note that hardcoding or directly passing private keys should never be done in a development or production environment.

▶ Here are some options for securely storing and retrieving private keys ↓

Regardless of how a private key is stored and retrieved, it's important to keep it secure and not expose it unnecessarily.

:::

<Tabs defaultValue="20" values={[{ label: 'ERC-20', value: '20', }, { label: 'ERC-721', value: '721', }, { label: 'ERC-1155', value: '1155', },] }>

This command deposits ERC-20 tokens from a rootchain to a Supernet.

- Replace `hex_encoded_depositor_private_key` with the private key of the account that will be depositing the tokens.
- Replace `receivers_addresses` with a comma-separated list of Ethereum addresses that will receive the tokens.
- Replace `amounts` with a comma-separated list of token amounts to be deposited for each receiver.
- Replace `root_erc20_token_address` with the address of the ERC-20 token contract on the rootchain.
- Replace `root_erc20_predicate_address` with the address of the ERC-20 predicate contract on the rootchain.
- Replace `root_chain_json_rpc_endpoint` with the JSON-RPC endpoint of the rootchain.
- Replace `minter-key` with the private key of the minter account. If the private key is provided, tokens are minted to the sender account prior to depositing them.

```
./polygon-edge bridge deposit-erc20 \
    --sender-key <hex_encoded_depositor_private_key> \
    --receivers <receivers_addresses> \
    --amounts <amounts> \
    --root-token <root_erc20_token_address> \
    --root-predicate <root_erc20_predicate_address> \
    --json-rpc <root_chain_json_rpc_endpoint>
    --minter-key <minter account hex encoded private key>
```

▶ Example ↓

This command deposits ERC-721 tokens from a rootchain to a Supernet.

- Replace `hex_encoded_depositor_private_key` with the private key of the account that will be depositing the tokens.
- Replace `receivers_addresses` with a comma-separated list of Ethereum addresses that will receive the tokens.
- Replace `token_ids` with a comma-separated list of token IDs that will be sent to the receivers' accounts.
- Replace `root_erc721_token_address` with the address of the ERC-721 token contract on the rootchain.
- Replace `root_erc721_predicate_address` with the address of the ERC-721 predicate contract on the rootchain.
- Replace `root_chain_json_rpc_endpoint` with the JSON-RPC endpoint of the rootchain.
- Replace `minter-key` with the private key of the minter account. If the private key is provided, tokens are minted to the sender account prior to depositing them.

```
./polygon-edge bridge deposit-erc721 \
    --sender-key <hex_encoded_depositor_private_key> \
    --receivers <receivers_addresses> \
    --token-ids <token_ids> \
    --root-token <root_erc721_token_address> \
    --root-predicate <root_erc721_predicate_address> \
```

```
    --json-rpc <root_chain_json_rpc_endpoint>
    --minter-key <minter account hex encoded private key>
```

▶ Example ↓

This command deposits ERC-1155 tokens from the rootchain to the Supernet.

- Replace `depositor_private_key` with the private key of the account that will be depositing the tokens.
- Replace `receivers_addresses` with a comma-separated list of Ethereum addresses that will receive the tokens.
- Replace `amounts` with a comma-separated list of token amounts to be deposited for each receiver.
- Replace `token_ids` with a comma-separated list of token IDs to be deposited.
- Replace `root_erc1155_token_address` with the address of the ERC-1155 token contract on the rootchain.
- Replace `root_erc1155_predicate_address` with the address of the ERC-1155 predicate contract on the rootchain.
- Replace `root_chain_json_rpc_endpoint` with the JSON-RPC endpoint of the rootchain.
- Replace `minter-key` with the private key of the minter account. If the private key is provided, tokens are minted to the sender account prior to depositing them.

```
./polygon-edge bridge deposit-erc1155 \
    --sender-key <depositor_private_key> \
    --receivers <receivers_addresses> \
    --amounts <amounts> \
    --token-ids <token_ids> \
    --root-token <root_erc1155_token_address> \
    --root-predicate <root_erc1155_predicate_address> \
    --json-rpc <root_chain_json_rpc_endpoint>
    --minter-key <minter account hex encoded private key>
```

▶ Example ↓

--- id: supernets-cross-chain-withdraw title: How To Perform Cross-Chain Asset Transfers From Childchain to Rootchain sidebar_label: How to Withdraw (Childchain → Rootchain) description: "A guide on the available bridge transactions." keywords: - docs - polygon - edge - supernets - bridge - crosschain - transactions - how to - how-to ---

import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem';

Prerequisites

You'll need to have a a successful bridge deployment to make any cross-chain transactions. If you haven't done so already, check out the local deployment guide here.

:::caution Key management and secure values When passing values to run transactions, it is important to keep sensitive values like private keys and API keys secure.

The sample commands provided in this guide use sample private keys for demonstration purposes only, in order to show the format and expected value of the parameter. It is important to note that hardcoding or directly passing private keys should never be done in a development or production environment.

▶ Here are some options for securely storing and retrieving private keys ↓

Regardless of how a private key is stored and retrieved, it's important to keep it secure and not expose it unnecessarily.

:::

<Tabs defaultValue="20" values={[{ label: 'ERC-20', value: '20', }, { label: 'ERC-721', value: '721', }, { label: 'ERC-1155', value: '1155', },] }>

Withdraw

This command withdraws ERC-20 tokens from a Supernet to a rootchain.

- Replace `hex_encoded_txn_sender_private_key` with the private key of the account that will be sending the transaction.
- Replace `receivers_addresses` with a comma-separated list of Ethereum addresses that will receive the tokens.
- Replace `amounts` with a comma-separated list of token amounts to be withdrawn for each receiver.
- Replace `child_erc20_predicate_address` with the address of the ERC-20 predicate contract on the Supernet.
- Replace `child_erc20_token_address` with the address of the ERC-20 token contract on the Supernet (optional, only required if the token is not a default ERC-20 token).
- Replace `child_chain_json_rpc_endpoint` with the JSON-RPC endpoint of the Supernet.

```
./polygon-edge bridge withdraw-erc20 \
    --sender-key <hex_encoded_txn_sender_private_key> \
    --receivers <receivers_addresses> \
    --amounts <amounts> \
    --child-predicate <rchild_erc20_predicate_address> \
    --child-token <child_erc20_token_address> \
    --json-rpc <child_chain_json_rpc_endpoint>
```

▶ Example ↓

Exit

This command sends an exit transaction to the `ExitHelper` contract on the rootchain for a token that was deposited on the Supernet. It basically finalizes a withdrawal (initiated on the Supernets) and transfers assets to receiving address on a rootchain.

- Replace `hex_encoded_txn_sender_private_key` with the private key of the account that will send the exit transaction.
- Replace `exit_helper_address` with the address of the `ExitHelper` contract on the rootchain.
- Replace `exit_id` with the ID of the exit event.
- Replace `root_chain_json_rpc_endpoint` with the JSON-RPC endpoint of the rootchain.
- Replace `child_chain_json_rpc_endpoint` with the JSON-RPC endpoint of the Supernet.

```
./polygon-edge bridge exit \
    --sender-key <hex_encoded_txn_sender_private_key> \
    --exit-helper <exit_helper_address> \
    --exit-id <exit_event_id> \
    --root-json-rpc <root_chain_json_rpc_endpoint> \
    --child-json-rpc <child_chain_json_rpc_endpoint>
```

▶ Example ↓

Withdraw

This command withdraws ERC-721 tokens from a Supernet to a rootchain.

- Replace `hex_encoded_sender_private_key` with the private key of the account that will initiate the withdrawal.
- Replace `receivers_addresses` with a comma-separated list of Ethereum addresses that will receive the withdrawn tokens on the rootchain.
- Replace `token_ids` with a comma-separated list of token IDs to be withdrawn.
- Replace `child_predicate_address` with the address of the predicate contract on the Supernet that holds the tokens being withdrawn.
- Replace `child_token_address` with the address of the ERC-721 token contract on the Supernet that holds the tokens being withdrawn.
- Replace `child_chain_json_rpc_endpoint` with the JSON-RPC endpoint of the Supernet.

```
./polygon-edge bridge withdraw-erc721 \
    --sender-key <hex_encoded_sender_private_key> \
    --receivers <receivers_addresses> \
    --token-ids <token_ids> \
    --child-predicate <child_predicate_address> \
    --child-token <child_token_address> \
    --jsonrpc <child_chain_json_rpc_endpoint>
```

▶ Example ↓

Exit

This command sends an exit transaction to the `ExitHelper` contract on the rootchain for a token that was deposited on the Supernet. It basically finalizes a withdrawal (initiated on the Supernets) and transfers assets to receiving address on a rootchain.

- Replace `hex_encoded_txn_sender_private_key` with the private key of the account that will send the exit transaction.
- Replace `exit_helper_address` with the address of the `ExitHelper` contract on the rootchain.
- Replace `exit_id` with the ID of the exit event.
- Replace `root_chain_json_rpc_endpoint` with the JSON-RPC endpoint of the rootchain.
- Replace `child_chain_json_rpc_endpoint` with the JSON-RPC endpoint of the Supernet.

```
./polygon-edge bridge exit \
    --sender-key <hex_encoded_txn_sender_private_key> \
    --exit-helper <exit_helper_address> \
    --exit-id <exit_event_id> \
    --root-json-rpc <root_chain_json_rpc_endpoint> \
    --child-json-rpc <child_chain_json_rpc_endpoint>
```

▶ Example ↓

Withdraw

This command withdraws ERC-1155 tokens from the Supernet to the rootchain.

- Replace `hex_encoded_withdrawer_private_key` with the private key of the account that will be withdrawing the tokens.
- Replace `receivers_addresses` with a comma-separated list of Ethereum addresses that will receive the tokens.
- Replace `amounts` with a comma-separated list of token amounts to be withdrawn for each receiver.
- Replace `token_ids` with a comma-separated list of token IDs to be withdrawn.
- Replace `root_erc1155_token_address` with the address of the ERC-1155 token contract on the rootchain.
- Replace `root_erc1155_predicate_address` with the address of the ERC-1155 predicate contract on the rootchain.
- Replace `child_chain_json_rpc_endpoint` with the JSON-RPC endpoint of the Supernet.

```
./polygon-edge bridge withdraw-erc1155 \
    --sender-key <hex_encoded_withdrawer_private_key> \
    --receivers <receivers_addresses> \
    --amounts <amounts> \
```

```
    --token-ids <token_ids> \
    --root-token <root_erc1155_token_address> \
    --root-predicate <root_erc1155_predicate_address> \
    --json-rpc <child_chain_json_rpc_endpoint>
```

▶ Example ↓

Exit

This command sends an exit transaction to the `ExitHelper` contract on the rootchain for a token that was deposited on the Supernet. It basically finalizes a withdrawal (initiated on the Supernets) and transfers assets to receiving address on a rootchain.

- Replace `hex_encoded_txn_sender_private_key` with the private key of the account that will send the exit transaction.
- Replace `exit_helper_address` with the address of the `ExitHelper` contract on the rootchain.
- Replace `exit_id` with the ID of the exit event.
- Replace `root_chain_json_rpc_endpoint` with the JSON-RPC endpoint of the rootchain.
- Replace `child_chain_json_rpc_endpoint` with the JSON-RPC endpoint of the Supernet.

```
./polygon-edge bridge exit \
    --sender-key <hex_encoded_txn_sender_private_key> \
    --exit-helper <exit_helper_address> \
    --exit-id <exit_event_id> \
    --root-json-rpc <root_chain_json_rpc_endpoint> \
    --child-json-rpc <child_chain_json_rpc_endpoint>
```

▶ Example ↓

--- id: supernets-ibft-to-polybft title: State Trie Migration and Regenesis to PolyBFT sidebar_label: Migrate from Edge PoA description: "Transition to a New Blockchain with PolyBFT Consensus from a Blockchain using IBFT Consensus with Edge PoA." keywords: - docs - Polygon - edge - supernets - childchain - network - modular ---

Transitioning to a New Blockchain with PolyBFT Consensus from a Blockchain using IBFT Consensus with Edge PoA

In this guide, you'll discover how to migrate from an existing IBFT consensus chain on the former Edge client to a new PolyBFT consensus-based chain.

:::caution This is not an upgrade

The regenesis process is not intended as an upgrade to another version of the consensus client but rather as a transformation to a distinct product suite designed for next-generation, application-specific chains (appchains) with cross-chain compatibility and staking requirements. If you are unfamiliar or uncertain about this, please refer to the introductory material and system design documentation. For further assistance, please reach out to the Supernets team. :::

:::info Compatibility The regenesis process has ONLY been tested for version(s) 0.6.x and not with earlier versions of the former Edge client. The team is actively working on testing compatibility with older versions. For assistance, please reach out to the Supernets team.

In general, it is recommended that you operate on the latest version of the old client. The latest version of the former Edge consensus client is v0.6.3. To upgrade Edge, please refer to the archived Edge documentation, available **here**.

:::

Overview

The regenesis process focuses on the migration of the state trie, also known as the state tree, from an existing chain to a new chain. The state trie is a Merkle Patricia trie that maintains the entire state of the blockchain, including account balances, contract storage, and other relevant data. Regenesis also involves processing block headers and validating state roots to ensure the correctness and consistency of the migrated state trie. However, regenesis does not involve the migration of transaction history, as it is specifically concerned with state trie migration and block header validation.

What you'll learn

In this guide, you'll gain an understanding of the following concepts:

- Migrating from an existing IBFT consensus chain to a new chain using the PolyBFT consensus algorithm.
- Generating a trie snapshot, setting up new validators, and creating the chain configuration for the new chain.
- Launching a new PolyBFT chain and ensuring the successful transfer of the snapshot trie to each validator node.

What you'll do

- **State Trie Migration**: The process starts with obtaining the trie root of the existing chain. A snapshot of the trie is created using this root, which will be utilized in the new chain.

- **Chain Setup**: The old chain data is removed, and new validators are created for the new chain. The genesis file is generated for the new chain, using the trie root from the snapshot.

- **Starting the New Chain**: The new PolyBFT chain is started, and the snapshot trie is copied to each validator node's data directory. This process ensures that the new chain's state is consistent with the old chain's state.

- **Resuming the Chain**: The new chain is restarted, and the validator nodes continue to seal new blocks. The state trie from the previous chain has been successfully migrated, allowing the new chain to maintain the same account balances, contract storage, and other state data.

- **Verification**: After the regenesis process is complete, you can verify the success of the migration by checking the account balances on the new chain. If the account balance is non-zero and matches the previous chain's state, the snapshot import was successful.

Before beginning the regenesis process, there are some prerequisites that must be fulfilled.

Prerequisites

- Set up a local IBFT consensus cluster for development purposes.

  A dedicated script is provided as part of the client to facilitate the cluster setup, encompassing key generation, network configuration, data directory creation, and genesis block generation. To create an IBFT cluster, execute the following command:

  ```
  scripts/cluster ibft
  ```

- Ensure that the accounts have sufficient funds.

  Before performing the regenesis, it's essential to ensure that the accounts have sufficient funds. You can check the balance using the following command:

  ```
  curl -s -X POST --data '{"jsonrpc":"2.0", "method":"eth_getBalance", "params":
  ["0x85da99c8a7c2c95964c8efd687e95e632fc533d6", "latest"], "id":1}' http://localhost:10002
  ```

  :::caution Don't use the develop branch for deployments

  Please ensure that you are not running on the `develop` branch, which is the active development branch and include changes that are still being tested and not compatible with the current process.

  Instead, use the latest release for deployments.

  :::

Regenesis

The regenesis process involves several steps, which are outlined below.

:::note Configuration Parameters

For a list of configurable parameters, please refer to the reference guide available here.

:::

1. Get trie root

Obtain the trie root of the existing chain, which is required to create a snapshot of the state trie.

```
./polygon-edge regenesis getroot --rpc "http://localhost:10002"
```

> *`10002` is the port number that is specified for the HTTP-RPC server in the configuration file for the polygon-edge node. It is the default port number used by the polygon-edge client for the HTTP-RPC server.*

2. Create trie snapshot*

Create a snapshot of the state trie using the trie root. This snapshot contains the state data, including account balances and contract storage. The `stateRoot` is the root hash of the Merkle trie data structure that represents the current state of the blockchain at a given block height. It can be obtained by querying the blockchain node through its JSON-RPC API.

```
./polygon-edge regenesis \
--target-path ./trie_new \
--stateRoot 0xf5ef1a28c82226effb90f4465180ec3469226747818579673f4be929f1cd8663 \
--source-path ./test-chain-1/trie
```

3. Remove old chain data

Delete the data of the old chain in preparation for setting up the new chain.

```
rm -rf test-chain-*
```

4. Create new validators

Generate new validators for the new chain, ensuring that the chain operates with a fresh set of validator nodes.

> *Please note that the `--insecure` flag used in the command is for testing purposes only and should not be used in production environments. Instead, secure methods should be used to store and pass secrets.*

```
./polygon-edge polybft-secrets --insecure --data-dir test-chain- --num 4
```

5. Generate the genesis file

Generate the genesis file for the new chain, using the trie root from the snapshot. This file is essential for starting the new chain with the migrated state data.

```
./polygon-edge genesis --consensus polybft --bridge-json-rpc http://127.0.0.1:8545 \
--block-gas-limit 10000000 \
--epoch-size 10 \
--trieroot 0xf5ef1a28c82226effb90f4465180ec3469226747818579673f4be929f1cd8663
```

6. Start your new PolyBFT chain

Launch the new PolyBFT chain with the copied snapshot trie, ensuring that the new chain's state is consistent with the old chain's state.

```
./polygon-edge server --data-dir ./test-chain-1 --chain genesis.json --grpc-address :10000 --libp2p :30301 --jsonrpc :10002 --
log-level DEBUG &
./polygon-edge server --data-dir ./test-chain-2 --chain genesis.json --grpc-address :20000 --libp2p :30302 --jsonrpc :20002 --
log-level DEBUG &
./polygon-edge server --data-dir ./test-chain-3 --chain genesis.json --grpc-address :30000 --libp2p :30303 --jsonrpc :30002 --
log-level DEBUG &
./polygon-edge server --data-dir ./test-chain-4 --chain genesis.json --grpc-address :40000 --libp2p :30304 --jsonrpc :40002 --
log-level DEBUG &
```

7. Copy the snapshot trie to the data directory

Copy the snapshot trie to the data directories of each validator node, allowing the new chain to maintain the same account balances, contract storage, and other state data.

```
rm -rf ./test-chain-1/trie
rm -rf ./test-chain-2/trie
rm -rf ./test-chain-3/trie
rm -rf ./test-chain-4/trie
cp -fR ./trie_new/ ./test-chain-1/trie/
cp -fR ./trie_new/ ./test-chain-2/trie/
cp -fR ./trie_new/ ./test-chain-3/trie/
cp -fR ./trie_new/ ./test-chain-4/trie/
```

8. Re-run the chain

Restart the new chain, and the validator nodes will continue to seal new blocks. The state trie from the previous chain has been successfully migrated.

```
./polygon-edge server --data-dir ./test-chain-1 --chain genesis.json --grpc-address :10000 --libp2p :30301 --jsonrpc :10002 --
seal --log-level DEBUG &
./polygon-edge server --data-dir ./test-chain-2 --chain genesis.json --grpc-address :20000 --libp2p :30302 --jsonrpc :20002 --
seal --log-level DEBUG &
./polygon-edge server --data-dir ./test-chain-3 --chain genesis.json --grpc-address :30000 --libp2p :30303 --jsonrpc :30002 --
seal --log-level DEBUG &
./polygon-edge server --data-dir ./test-chain-4 --chain genesis.json --grpc-address :40000 --libp2p :30304 --jsonrpc :40002 --
seal --log-level DEBUG &
```

9. Check account balance on the new PolyBFT chain

Verify the success of the migration by checking the account balances on the new chain. If the account balance is non-zero and matches the previous chain's state, the snapshot import was successful.

```
curl -s -X POST --data '{"jsonrpc":"2.0", "method":"eth_getBalance", "params":["0x85da99c8a7c2c95964c8efd687e95e632fc533d6",
"latest"], "id":1}' http://localhost:10002

{"jsonrpc":"2.0","id":1,"result":"0x3635c9adc5dea00000"}%
```

---

id: supernets-install title: Installation sidebar_label: Install Binaries description: "Steps on how to install the Supernet binaries." keywords: - docs - polygon - edge - install - installation

import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem'; import DownloadButton from '@site/src/data/DownloadButton';

:::tip

We recommend using the pre-built releases and verifying the provided checksums for security.

The Docker image is also a convenient option for containerized deployment. Building from source provides greater flexibility, but requires a suitable development environment.

:::

<Tabs defaultValue="pre-built" values={[{ label: 'Pre-built releases', value: 'pre-built', }, { label: 'Docker image', value: 'docker', }, { label: 'Build from source', value: 'source', },] }>
To access the pre-built releases, visit the GitHub releases page. The client provides cross-compiled AMD64/ARM64 binaries for Darwin and Linux.

:::info Latest release: 1.0.0

The latest stable test release is v1.0.0.

:::

To use Docker, you will need to have it installed on your system. If you haven't already installed Docker, you can follow the instructions on the official Docker website for your operating system.

You can access the official Polygon Edge Docker images hosted under the 0xpolygon registry using the following command:

```
docker pull 0xpolygon/polygon-edge:latest
```

> *Before getting started, ensure you have Go installed on your system (version >= 1.15 and <= 1.19). Compatibility is being worked on for other versions and will be available in the near future.*

> *To install Go, run the following command in your CLI (we are using 1.18 in this example): `sudo apt-get install golang-1.18`. Or, use a package manager like Snapcraft for Linux, Homebrew for Mac, and Chocolatey for Windows.*

Use the following commands to clone the Polygon Edge repository and build from source:

```
git clone https://github.com/0xPolygon/polygon-edge.git
cd polygon-edge/
go build -o polygon-edge .
```

--- id: supernets-param-reference title: Configuration Parameter Reference sidebar_label: Configuration Parameter Reference description: "Manpage for configurable parameters." keywords: - docs - polygon - Supernets - flags - parameters ---
import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem';

Configuration parameters are crucial for setting up and operating a Supernet. You can configure these parameters using the genesis and server commands. Before running these commands, it is essential to generate keys using the polybft-secrets command.

For information on the available configuration flags and their descriptions for customizing the genesis configuration of a Supernet, refer to the categorized tabs below.

<Tabs defaultValue="polybft" values={[{ label: 'PolyBFT', value: 'polybft', }, { label: 'Genesis', value: 'genesis', }, { label: 'Server', value: 'server', },] }>

PolyBFT Configuration Parameter Reference

Parameter	Description	Default Value	Mandatory
`--account`	The flag indicating whether a new account is created	TRUE	NO
`--config string`	The path to the SecretsManager config file	""	NO
`--data-dir string`	The directory for the Polygon Edge data if the local FS is used	""	YES
`--insecure`	The flag indicating whether the secrets stored locally are encrypted	FALSE	NO
`--network`	The flag indicating whether a new Network key is created	TRUE	NO
`--num int`	The number of secrets to be created (only for local FS)	1	NO
`--output`	The flag indicating whether to output existing secrets	FALSE	NO
`--private`	The flag indicating whether the private key is printed	FALSE	NO

:::info Mutually Exclusive Paramaters

- `--config` and `--data-dir`: These flags are mutually exclusive. Use either `--config` to specify the path to the SecretsManager config file or `--data-dir` to set the directory for the Polygon Edge data if the local FS is used.
- `--num` and `--config`: These flags are mutually exclusive. Set `--num` to define the number of secrets to be created (only for local FS) or use `--config` to provide the SecretsManager config file path.

:::

Genesis Configuration Parameter Reference

Parameter	Description	Default Value	Mandatory	Examp
`--block-gas-limit`	The maximum amount of gas used by all transactions in a block	5242880	NO	`genesis`
`--block-time`	The predefined period which determines block creation frequency	2s	NO	`genesis`
`--block-time-drift`	Configuration for block time drift value (in seconds). Defines the time slot in which a new block can be created	10	NO	`genesis`
`--bootnode`	MultiAddr URL for p2p discovery bootstrap. This flag can be used multiple times.	N/A	NO	`genesis` `"/ip4/1`
`--bridge-allow-list-admin`	List of addresses to use as admin accounts in the bridge allow list.	[]string{}	NO	`genesis`
`--bridge-allow-list-enabled`	List of addresses to enable by default in the bridge allow list.	[]string{}	NO	`genesis`
`--burn-contract`	The burn contract blocks and addresses (format: [block]:[address])	[]string{}	NO	`genesis`
`--consensus`	The consensus protocol to be used	"polybft"	NO	`genesis`
`--dir`	Represents the file path for the genesis data	"./genesis.json"	NO	`genesis`
`--epoch-reward`	Reward size for block sealing	1	NO	`genesis`
`--epoch-size`	The epoch size for the chain	100000	NO	`genesis`
`--name`	The name for the chain	"polygon-edge"	NO	`genesis`
`--premine`	The premined accounts and balances	[]string{}	NO	`genesis`
`--sprint-size`	The number of blocks included into a sprint	5	NO	`genesis`
`--trieroot`	Trie root from the corresponding triedb	""	NO	`genesis`
`--validators`	Initial validator addresses for the chain	[]string{}	YES	`genesis`
`--validators-path`	Root path containing polybft validators' secrets	"./"	NO	`genesis`
`--validators-secret`	Validators secrets	[]string{}	NO	`genesis` `"0x0101`
`--artifacts-path`	The path to the contract artifacts JSON	""	YES	`genesis`
`--chain`	The genesis file to update	"./genesis.json"	NO	`genesis`
`--constructor-args`	The constructor arguments, if any	""	NO	`genesis`
`--predeploy-address`	The address to predeploy to. Must be >= 0x0000000000000000000000000000000000001100	"0x0000000000000000000000000000000000001100"	YES	`genesis`

:::info Mutually Exclusive Paramaters

- `--validators` : Validators defined by the user (format: `<P2P multi address>:<public ECDSA address>:<public BLS key>`). If this flag is set, the entire multi address must be specified. If not set, validators configuration will be read from `--validators-path` .
- `--validators-path` : Root path containing polybft validators' secrets. If `--validators` flag is not specified, validators' configuration will be read from this path.
- `--validators-prefix` : Folder prefix names for polybft validators' secrets. If `--validators` flag is set, this prefix will be used for folder names.

:::

Server Configuration Parameter Reference

Parameter	Description	Default Value	Mandatory	Example	Reconfigurable at Runtime
`--grpc-address` string	The address of the GRPC interface.	"127.0.0.1:9632"	NO	Command: server Flag: --grpc-address “0.0.0.0:10000”	NO
`--jsonrpc` string	The address of the JSON-RPC interface.	"0.0.0.0:8545"	NO	Command: server Flag: --jsonrpc “0.0.0.0:10002”	NO
`--log-level` string	The log level for the console output.	“INFO”	NO	Command: server Flag: --log-level “DEBUG”	NO
`--chain` string	The genesis file used for starting the chain. The genesis file is generated by running the genesis CLI command.	"./genesis.json"	NO	Command: server Flag: --chain “genesis.json”	NO
`--config` string	The path to the CLI config. Supported extensions are: .json, .hcl, .yaml and .yml. If this flag is set, other flags will be overridden. If some value that will be overridden is not specified in a config file, default value for that parameter is used.	“”	NO	Command: server Flag: --config “config.json”	NO
`--data-dir` string	The data directory used for storing Polygon Edge client data.	“”	YES	Command: server Flag:--data-dir “./test-chain-1”	NO
`--libp2p` string	The address and port for the libp2p service.	“127.0.0.1:1478”	NO	Command: server Flag: --libp2p “0.0.0.0:30301”	NO
`--prometheus` string	The address and port for the prometheus instrumentation service (address:port). If only port is defined (:port) it will bind to 0.0.0.0:port.	“”	NO	Command: server Flag: --prometheus “0.0.0.0:5001”	NO
`--nat` string	The external IP address without port, as can be seen by peers. The string specidied can be in IPv4 dotted decimal ("192.0.2.1"), IPv6 ("2001:db8::68"), or IPv4-mapped IPv6 ("::ffff:192.0.2.1") form.	“”	NO	Command: server Flag:--nat "192.0.2.1"	NO
`--dns` string	The host DNS address which can be used by a remote peer for connection.	“”	NO	Command: server Flag: --dns "www.example.com"	NO
`--block-gas-target` string	The target block gas limit for the chain. If omitted, the value of the parent block is used which will be the value set by the `--block-gas-limit` flag of the genesis command. If this flag is set, the block fill take block gas limit of the parent block and increment it by small delta (parentGasLimit /1024). If the block gas target is reached that the value of it will be set as a gas limit for the current block.	0x0	NO	Command: server Flag: --block-gas-target “10000000”	YES, this parameter can be changed by stopping the node and then starting it again with the server command and specifying --block-gas-target flag providing the new value e.g. --block-gas-target “60000000”
`--secrets-config` string	The path to the SecretsManager config file. Used for Hashicorp Vault. If omitted, the local FS secrets manager is used.	“”	NO	Command: server Flag: --secret-config “hashicorp.json”	NO
`--restore` string	The path to the archive blockchain data to restore on initialization.	“”	NO	Command: server Flag: --restore	NO
`--seal`	The flag indicating that the client should seal blocks.	TRUE	NO	Command: server Flag: --seal	NO
`--no-discover`	Prevent the client from discovering other peers.	FALSE	NO	Command: server Flag: --no-discover	NO
`--max-peers` int	The client's max number of peers allowed.	40	NO	Command: server Flag: --max-peers “70”	NO
`--max-inbound-peers` int	The client's max number of inbound peers allowed.	32	NO	Command: server Flag:--max-inbound-peers “50”	NO

`--max-outbound-peers` int	The client's max number of outbound peers allowed.	8	NO	Command: server Flag: --max-outbound-peers “20”	NO
`--price-limit` uint	The minimum gas price limit to enforce for acceptance into the pool.	0	NO	Command: server Flag: --price-limit “1”	YES, this parameter can be changed by stopping the node and then starting it again with the server command and specifying --price-limit flag providing the new value e.g. --price-limit “5”
`--max-slots` uint	Maximum slots in the transaction pool. When the maximum capacity is reached, transaction is not stored in the pool. One transaction occupies txSize/32kB number of slots. If e.g. --max-slots is 5, and there are tx1 which has 2kB and tx2 which has 33kB, that means that 3 slots are occupied and there are 2 free slots left. This parameter refers to the enqueued and promoted transactions in the pool.	4096	NO	Command: server Flag: --max-slots “100000”	NO
`--max-enqueued` uint	Maximum number of enqueued transactions in the pool per account.	128	NO	Command: server Flag: --max-enqueued “200”	NO
`--access-control-allow-origins` stringArray	The CORS(cross origin resource sharing) header indicating whether any JSON-RPC response can be shared with the specified origin.	[]string{"*"}	NO	Command: server Flag: --access-control-allow-origins “https://foo.example”	NO
`--json-rpc-batch-request-limit` uint	Max length to be considered when handling json-rpc batch requests, value of 0 disables it.	20	NO	Command: server Flag: --json-rpc-batch-request-limit	NO
`--json-rpc-block-range-limit` uint	Max block range to be considered when executing json-rpc requests that consider fromBlock/toBlock values (e.g. eth_getLogs), value of 0 disables it.	1000	NO	Command: server Flag: --json-rpc-block-range-limit “2000”	NO
`--log-to` string	Write all logs to the file at specified location instead of writing them to console.	“”	NO	Command: server Flag: --log-to “edge-log.log”	NO
`--relayer`	Start the state sync relayer service.	FALSE	NO	Command: server Flag: --relayer	NO
`--num-block-confirmations` uint	Minimal number of child blocks required for the parent block to be considered final. This parameter is used by the event Tracker when reading logs from the parent chain.	64	NO	Command: server Flag: --num-block-confirmations “2”	NO

:::info Mutually Exclusive Paramaters

- `--max-inbound-peers` : The client's maximum number of inbound peers allowed. Default value is 32.
- `--max-outbound-peers` : The client's maximum number of outbound peers allowed. Default value is 8.

:::
--- id: supernets-performance title: Performance Benchmarks sidebar_label: Performance Benchmarks description: "The first performance benchmarks for Supernets." keywords: - docs - polygon - edge - Supernets - metrics - performance ---
This document presents a summary of performance benchmarks for Supernets v1.0 release candidate (v1.0-rc1).

Summary of Test Results

Transaction Type	TPS Sent	TPS Mined	Gas per Second	Gas per Transaction	Instance Type(s)
EOA to EOA	2,600	2,442	51,282,000	21,000	x2iezn.2xlarge
ERC20	1,300	591	21,696,201	36,711	x2iezn.2xlarge
ERC721	800	558	32,642,442	58,499	x2iezn.2xlarge

Test Environment

The performance tests were conducted in a controlled environment to ensure accuracy and consistency in the results. The environment setup includes the following components:

Network Configuration

- Network: Polygon Edge
- Consensus Algorithm: PolyBFT (v1.0-rc1)

Instance Types

The tests utilized various Amazon Web Services (AWS) instance types to evaluate the impact of varying compute resources on network performance:

- **t2.xlarge**
- **t2.micro**
- **c5.2xlarge**
- **c6a.48xlarge**
- **x2iezn.2xlarge**
- **c6a.xlarge**

▶ General instance specifications

Transaction Types

The tests were performed using three different types of transactions to assess the network's capability to handle various use cases:

- **EOA to EOA**: Representing simple value transfers between user accounts.
- **ERC20**: Token transfers, simulating the exchange of tokens on the network.
- **ERC721**: Non-fungible token (NFT) transfers representing the exchange of unique digital assets.

Performance Metrics

The following key performance metrics were considered during the tests:

- **Transactions per second (TPS) sent**: The number of transactions submitted to the network per second.
- **Transactions per second (TPS) mined**: The number of transactions successfully processed and included in the blockchain per second.
- **Gas per transaction**: The amount of gas consumed by each transaction, representing the computational cost.
- **Gas per second**: The total amount consumed per second indicates the network's overall computational capacity.

---

id: supernets-quick-start title: Quick Start sidebar_label: Quick Start description: "Spin up a new childchain instance with one-click." keywords: - docs - polygon - edge - supernets - quick - deploy - cluster

import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem'; import DownloadButton from '@site/src/data/DownloadButton';

Before proceeding, ensure that your system meets the necessary system requirements.

Spawn a local Supernet in 2 minutes

To access the pre-built releases, visit the GitHub releases page. The client provides cross-compiled AMD64/ARM64 binaries for Darwin and Linux.

To run the Supernets test environment locally, run the following command from the project's root:

```
./scripts/cluster polybft
```

That's it! You should have successfully been able to start a local Supernet just by running the script.

> - *Stop the network: "CTRL/Command C" or* `./scripts/cluster polybft stop`.
> - *Destroy the network:* `./scripts/cluster polybft destroy`.

▶ Deployment script details ↓

--- id: supernets-requirements title: System Requirements sidebar_label: System Requirements description: "System requirements to successfully run Supernets on a local or remote machine." keywords: - docs - polygon - edge - install - installation ---

Minimum Hardware Configuration

:::tip Hardware environment tips

While we do not favor any operating system, more secure and stable Linux server distributions (like CentOS) should be preferred over desktop operating systems (like Mac OS and Windows).

The minimum storage requirements will change over time as the network grows. It is recommended to use more than the minimum requirements to run a robust full node.

:::

This is the minimum hardware configuration required to set up a Supernet:

Component	Minimum Requirement	Recommended
Processor	4-core CPU	8-core CPU
Memory	8 GB RAM	16 GB RAM
Storage	200 GB SSD	1 TB SSD
Network	High-speed internet connection	Dedicated server with gigabit connection

> *Note that these minimum requirements are based on the x2iezn.2xlarge instance type used in the performance tests, which demonstrated satisfactory performance. However, for better performance and higher transaction throughput, consider using more powerful hardware configurations, such as those equivalent to x2iezn.4xlarge or x2iezn.8xlarge instance types.*

Prerequisites

Before starting any of the tutorials, you should understand the basics of blockchain technology and be familiar with command-line interfaces. It would help if you also had the `polygon-edge` binary installed on your machine. Check out the installation guide for more information if you haven't already.

Ensure you have the following system prerequisites:

Prerequisite	Description
Golang (v1.15-1.19)	Install Go using CLI or package manager like Snapcraft for Linux, Homebrew for Mac, or Chocolatey for Windows. Compatibility for other versions coming soon.
Docker	Required to run the geth instance. Follow official Docker documentation for installation instructions.
Internet connection	Stable internet connection required.
Network security	Ensure that network ports used by Polygon Edge are not blocked by firewalls or other security measures.
Operating system	Ensure that the latest security patches and updates are installed.

:::caution Solidity v0.8.19 or earlier recommended

Solidity v0.8.20 introduces new features, including the implementation of `PUSH0` opcode, which is not yet supported in Supernets. If you decide to use v0.8.20, ensure that you set your EVM version to "Paris" in the framework you use to deploy your contracts.

For now, we recommend using Solidity v0.8.19 or earlier.

:::

id: covalent title: Using Covalent sidebar_label: Covalent description: Learn how to use Covalent's unified API for data keywords: - docs - matic - polygon - covalent - data - analytics - index - indexing - query image: https://matic.network/banners/matic-network-16x9.png

:::caution Content disclaimer

Please view the third-party content disclaimer here.

:::

:::tip Quickstart

Check out **this introduction video** to get started.

:::

Supported endpoints

All **Class A** endpoints are supported for the Matic mainnet and the Mumbai testnet. You can query either network via the unified API by changing the `chainId`.

:::info Endpoints

A full list of all the requests you are able to do on the Polygon network using Covalent are available on the Covalent API documentation.

:::

Appendix

Matic Gas token

To interact with the Matic network, MATIC tokens are required to pay as gas fees. Covalent's responses automatically returns `gas_*` fields in the MATIC units.

Token mapping

Covalent maintains an on-chain real-time mapping of token addresses between the Ethereum mainnet and the Matic chain. These addresses are used to reverse-lookup prices on Matic and also to return the right token logo URLs.

Some example of mapped tokens:

Token	Ethereum mainnet	Matic mainnet
USDT	0xdac17f958d2ee523a2206206994597c13d831ec7	0xc2132d05d31c914a87c6611c10748aeb04b58e8f
Uniswap UNI	0x1f9840a85d5af5bf1d1762f925bdaddc4201f984	0xb33eaad8d922b1083446dc23f610c2567fb5180f

Token prices

For tokens that have a mapping back to Ethereum mainnet, Covalent is able to return the mapped prices.

id: graph-data title: Using subgraphs sidebar_label: Using Polygon's root subgraph description: Learn how to use Polygon's subgraph for data keywords: - docs - matic - polygon - the graph - data - analytics - index - indexing - query image: https://matic.network/banners/matic-network-16x9.png

:::caution Content disclaimer

Please view the third-party content disclaimer here.

:::

Polygon Root Subgraph Introduction

Polygon has a GraphQL API Endpoint hosted by The Graph called a subgraph for indexing and organizing data from the Polygon smart contracts.

This subgraph can be used to query on-chain Polygon data. The subgraph data is serviced by a decentralized group of server operators called Indexers.

The PolygonRoot subgraph works by listening for events emitted by one or more data sources (Smart Contracts) on the various chains. It handles the indexing and caching of data which can later be queried using the GraphQL API Endpoint, providing an excellent developer experience.

Helpful Resources

- Video Tutorial on creating an API Key
- Managing your API Key & setting your indexer preferences
- Querying from an application
- How to use the explorer and playground to query on-chain data
- Explorer Page
- Graphql Endpoint: https://gateway.thegraph.com/api/[api-key]/subgraphs/id/FDrqtqbp8LhG1hSnwtWB2hE6C97FWA54irrozjb2TtMH
- Code Repo

---

id: graph-entities title: Entity descriptions sidebar_label: Using Polygon's root subgraph description: Learn how to use Polygon's subgraph for data keywords: - docs - matic - polygon - the graph - data - analytics - index - indexing - query image: https://matic.network/banners/matic-network-16x9.png

:::caution Content disclaimer

Please view the third-party content disclaimer here.

:::

Entities

- Entities
  - Checkpoint
  - StateSync
  - StateRegistration
  - PredicateRegistration
  - TokenMapping
  - FxTokenMapping
  - FxTokenMappingCounter
  - FxDeposit
  - FxDepositCounter
  - FxWithdraw
  - FxWithdrawCounter
  - Validator
  - StakeUpdate
  - GlobalDelegatorCounter
  - Delegator
  - Topup
  - StakingNFTTransfer
  - DelegatorUnbond
  - MaticTransfer
  - GlobalDelegationCounter
  - Delegation

Checkpoint

Field	Type	Description
id	ID!	Checkpoint ID
Proposer	Bytes!	The address of the proposer

headerBlockId	BigInt!	The Id of the header block
checkpointNumber	BigInt!	The checkpoint number
reward	BigInt!	The reward of the proposed checkpoint
start	BigInt!	Start block of the proposed checkpoint
end	BigInt!	End block of the proposed checkpoint
root	Bytes!	Merkle root hash of the proposed checkpoint
logIndex	String!	LogIndex ID
transactionHash	Bytes!	Transaction hash of proposed checkpoint
timeStamp	BigInt!	The checkpoint timestamp

StateSync

Field	Type	Description
id	ID!	StateSync ID
stateId	BigInt!	The stateId number
contract	Bytes!	The contract address
syncType	Int!	Type of sync
depositorOrRootToken	String!	Depositor or root token address
depositedTokenOrChildToken	String!	Address of deposited token or child token address
data	String!	Data containing the state of the contract at sync time
rawData	String!	Raw data containing the state of the contract at sync time
logIndex	String!	logIndex ID
transactionHash	Bytes!	Transaction hash
timeStamp	BigInt!	Timestamp during sync of contract state
blockNumber	BigInt!	Block number of the state sync

StateRegistration

Field	Type	Description
id	ID!	State registration ID
user	Bytes!	User address
receiver	Bytes!	Receiver address
sender	Bytes!	Sender address

```
|
```

PredicateRegistration

Field	Type	Description
id	ID!	Predicate registration ID
tokenType	Bytes!	Token contract address
predicateAddress	Bytes!	Predicate address
timestamp	BigInt!	Predicate registration timestamp
transactionHash	Bytes!	Transaction hash of predicate registration

TokenMapping

Field	Type	Description
id	ID!	Token mapping ID
rootToken	Bytes!	Root token address
childToken	Bytes!	Child token address

tokenType	String!	Token contract address
isPOS	Boolean!	Checks wether POS is true or false
timestamp	BigInt!	Token mapping timestamp
transactionHash	Bytes!	Transaction hash

FxTokenMapping

Field	Type	Description
id	ID!	Fx token mapping ID
counter	BigInt!	Fx token mapping counter
contractAddress	Bytes!	Contract address that handles Fx token mapping
rootToken	Bytes!	Root token address
childToken	Bytes!	Child token address
tokenType	String!	Token contract address
timestamp	BigInt!	Fx token mapping timestamp
transactionHash	Bytes!	Fx transaction hash

FxTokenMappingCounter

Field	Type	Description
id	ID!	Fx token mapping ID
current	BigInt!	Current count for fx token mapping

FxDeposit

Field	Type	Description
id	ID!	Fx deposit ID
counter	BigInt!	Fx deposit counter
contractAddress	Bytes!	Contract address that handles fx deposits
rootToken	Bytes!	Root token address
tokenType	String!	Token contract address
depositor	Bytes!	Address of the depositor
userAddress	Bytes!	User address
amount	BigInt!	Amount of fx deposited in the transaction
tokenId	BigInt!	Token ID
timestamp	BigInt!	Block timestamp
transactionHash	Bytes!	Fx transaction hash

FxDepositCounter

Field	Type	Description
id	ID!	Fx deposit counter ID
current	BigInt!	Current number of FxDeposits

FxWithdraw

Field	Type	Description
id	ID!	Fx withdraw ID
counter	BigInt!	Fx withdraw counter
contractAddress	Bytes!	Contract address that handles withdrawals
rootToken	Bytes!	Root token address

childToken	Bytes!	Child token address
tokenType	String!	Token contract address
userAddress	Bytes!	User address
amount	BigInt!	Amount of fx withdrawn in the transaction
tokenId	BigInt!	Token ID
timestamp	BigInt!	Block timestamp
transactionHash	Bytes!	Fx transaction hash

FxWithdrawCounter

Field	Type	Description
id	ID!	Fx withdraw counter ID
current	BigInt!	Current number of FxWithdrawals

Validator

Field	Type	Description
id	ID!	Validator ID
validatorId	BigInt!	Validator ID
owner	Bytes!	Address of the owner
signer	Bytes!	Address of the signer
signerPubKey	Bytes!	Public key of the signer
liquidatedRewards	BigInt!	Liquidated reward for the validator
activationEpoch	BigInt!	Epoch validation was activated
deactivationEpoch	BigInt!	Epoch validation was deactivated
totalStaked	BigInt!	Total amount of tokens staked
selfStake	BigInt!	Amount staked by the validator
delegatedStake	BigInt!	Amount delegated to the validator for staking
commissionRate	BigInt!	Commission rate
nonce	BigInt!	Transaction nonce
status	Int!	Status codes: 0 - staked, 1 - unstaked, 2 - jailed, 3 - unjailed
jailEndEpoch	BigInt!	Epoch where validator is unjailed
auctionAmount	BigInt!	Auction amount
isInAuction	Boolean!	Checks wether validator is in auction

StakeUpdate

Field	Type	Description
id	ID!	Stake update ID
validatorId	BigInt!	Validator ID
totalStaked	BigInt!	Total amount staked by validator
block	BigInt!	Block number
nonce	BigInt!	Transaction nonce
transactionHash	Bytes!	Transaction hash
logIndex	BigInt!	Log index number

GlobalDelegatorCounter

Field	Type	Description
id	ID!	Global delegator counter ID

current	BigInt!	Keeps track of current delegator counter i.e. delegators are present as of now

Delegator

Field	Type	Description
id	ID!	Delegator ID
counter	BigInt!	Traverse through large number of delegator list
validatorId	BigInt!	Validator ID
address	Bytes!	Delegator address
delegatedAmount	BigInt!	Total delegated amount
unclaimedAmount	BigInt!	total unclaimed amount (after sellVoucher and before claiming it)
claimedAmount	BigInt!	total claimed amount (after withdraw delay, while claiming unstaked amount)
tokens	BigInt!	total current shares (works until tokens are non-transferable)
claimedRewards	BigInt!	Total claimed rewards

Topup

Field	Type	Description
id	ID!	Heimdall topup ID
address	Bytes!	Merkle root hash
topupAmount	BigInt!	Topup amount
withdrawAmount	BigInt!	Amount withdrawn

StakingNFTTransfer

Field	Type	Description
id	ID!	Staking NFT transfer ID
tokenId	BigInt!	NFT ID
currentOwner	Bytes!	Current owner address
previousOwners	[Bytes!]!	Array of previous owners addresses
transactionHashes	[Bytes!]!	Array of transaction hashes

DelegatorUnbond

Field	Type	Description
id	ID!	Delegator unbond ID
nonce	BigInt!	Transaction nonce
validatorId	BigInt!	Validator ID
user	Bytes!	Delegator address
amount	BigInt!	Total amount
tokens	BigInt!	Token amount
completed	Boolean!	Checks wether unbond is complete
unbondStartedTXHash	Bytes!	Transaction hash when unbond was initaited
unbondStartedTimeStamp	BigInt!	Timestamp when when unbond was initaited
unbondClaimedTXHash	Bytes!	Transaction hash when unbond was claimed
unbondClaimedTimeStamp	BigInt!	Timestamp when unbond was claimed
activeStake	BigInt!	Active stake

MaticTransfer

Field	Type	Description
id	ID!	Matic transfer ID

token	Bytes!	Token address
from	Bytes!	Sender address
to	Bytes!	Receiver address
value	BigInt!	Amount of matic transferred
block	BigInt!	Transaction block
timestamp	BigInt!	Transaction timestamp
transactionHash	Bytes!	Transaction hash

GlobalDelegationCounter

Field	Type	Description
id	ID!	Delegation counter ID
current	BigInt!	Current number of delegations

Delegation

Field	Type	Description
id	ID!	Delegation ID
counter	BigInt!	Traverse through large number of delegator list
validatorId	BigInt!	Validator ID
address	Bytes!	Delegator address
timestamp	BigInt!	Transaction timestamp
transactionHash	Bytes!	Delegation transaction hash
amount	BigInt!	Delegation amount
block	BigInt!	Transaction block
activeStake	BigInt!	Active stake

---

id: graph-queries title: Sample queries sidebar_label: Querying Polygon's root subgraph description: Learn how to use Polygon's subgraph for data keywords: - docs - matic - polygon - the graph - data - analytics - index - indexing - query image: https://matic.network/banners/matic-network-16x9.png

:::caution Content disclaimer

Please view the third-party content disclaimer here.

:::

Sample Queries

Below are some sample queries you can use to gather information from the polygon root subgraph

You can build your own queries using a GraphQL Explorer and enter your endpoint to limit the data to exactly what you need.

Checkpoints

Description: This query gets the first 20 checkpoints proposed and the validated blocks assigned to it as well as the rewards for each.

```
{
  checkpoints(orderBy: id, orderDirection: asc, first: 20) {
    id
    proposer
    start
    end
    checkpointNumber
    reward
    timestamp
    transactionHash
  }
}
```

Delegators

Description: This query gets the total number of active delegators, the first 50 delegations and the validator they delegated to.

```
{
  globalDelegatorCounters {
    id
    current
  }
  delegations(first: 50) {
    id
    block
    activeStake
    validatorId
  }
}
```

Transfers

Description: This query gets the first 50 matic token transfers, the sender, receiver, value, transaction block and timestamp.

```
{
  maticTransfers(first: 50, orderBy: id, orderDirection: asc) {
    id
    from
    to
    value
    block
    timestamp
  }
}
```

---

id: overview title: Setting up a Project with The Graph and Polygon PoS sidebar_label: Get Started description: Learn how to set up a hosted project with The Graph and Polygon. keywords: - polygon - matic - graph - the graph - index - query - subgraph image: https://matic.network/banners/matic-network-16x9.png slug: graph

import useBaseUrl from '@docusaurus/useBaseUrl';

:::caution Content disclaimer

Please view the third-party content disclaimer here.

:::

The Graph, a decentralized protocol for indexing and querying chain data, supports the Matic chain. Data defined through subgraphs is easy to query and explore. Subgraphs can be created locally, or use a free hosted explorer for indexing and data display.

> *Note: See https://thegraph.com/docs/quick-start for more details, local installation and more. The docs include an example for learning how subgraphs work and this video provides a good introduction.*

Steps

1. Go to Graph Explorer (https://thegraph.com/explorer/) and setup an account. You will need a GitHub account for authentication.
2. Go to your dashboard and click Add Subgraph. Define the subgraph Name, Account, and Subtitle and update the image and other info (you can update later) if desired.

<img src={useBaseUrl("img/graph/Graph-1.png")} width="100%" height="100%"/>

3. Install Graph CLI on your machine (using either npm or yarn)

```
$ npm install -g @graphprotocol/graph-cli
$ yarn global add @graphprotocol/graph-cli
```

4. The following command creates a subgraph that indexes all events of an existing contract. It attempts to fetch the contract ABI from BlockScout and falls back to requesting a local file path. If any of the optional arguments are missing, it takes you through an interactive form.

```
graph init \
  --from-contract <CONTRACT_ADDRESS> \
  [--network Matic ] \
  [--abi <FILE>] \
  <GITHUB_USER>/<SUBGRAPH_NAME> [<DIRECTORY>]

--network: choose "Matic" for Matic mainnet and "Mumbai" for Matic Testnet.
--from-contract <CONTRACT_ADDRESS> is the address of your existing contract which you have deployed on the Matic network:
Testnet or Mainnet.
```

```
--abi <FILE> is a local path to a contract ABI file (optional, If verified in BlockScout, the graph will grab the ABI, otherwise
you will need to manually add the ABI. You can save the abi from BlockScout or by running truffle compile or solc on a public
project.)
The <GITHUB_USER> is your github user or organization name, <SUBGRAPH_NAME> is the name for your subgraph, and <DIRECTORY> is
the optional name of the directory where graph init will put the example subgraph manifest.
```

> *Note: More details are here: https://thegraph.com/docs/define-a-subgraph#create-a-subgraph-project*

5. Authenticate with the hosted service

```
graph auth https://api.thegraph.com/deploy/ <your-access-token>
```

You can find the access token by going to your dashboard on the graph website.

6. cd to the directory you created and start defining the subgraph. Information on creating a subgraph is available in the Graph Docs here. https://thegraph.com/docs/define-a-subgraph
7. When you are ready, deploy your subgraph. You can always test and redeploy as needed.

> *If your previously deployed subgraph is still in status Syncing, it will be immediately replaced with the newly deployed version. If the previously deployed subgraph is already fully synced, Graph Node will mark the newly deployed version as the Pending Version, sync it in the background, and only replace the currently deployed version with the new one once syncing the new version has finished. This ensures that you have a subgraph to work with while the new version is syncing.*

```
yarn deploy
```

Your subgraph will be deployed and can be accessed from your dashboard.

You can learn about querying the subgraph here: https://thegraph.com/docs/query-the-graph#using-the-graph-explorer

If you want to make your subgraph public, you can do so by accessing your subgraph from your dashboard and then click on edit button. You will see the slider at the bottom of edit page.

id: hardhat title: How to Deploy a Smart Contract Using Hardhat sidebar_label: Get Started with Hardhat description: Use Hardhat to deploy a Smart Contract on Polygon keywords: - docs - matic - polygon - smart - contracts - hardhat - deploy on polygon image: https://wiki.polygon.technology/img/polygon-logo.png

Overview

Hardhat is an Ethereum development environment that provides an easy way to deploy smart contracts, run tests and debug Solidity code locally.

In this tutorial, you will learn how to set up Hardhat and use it to build, test and deploy a simple smart contract.

What you will do

- Set up Hardhat
- Create a simple smart contract
- Compile contract
- Test contract
- Deploy contract

Setting up the development environment

There are a few technical requirements before we start. Please install the following:

- Node.js v10+ LTS and npm (comes with Node)
- Git

Once we have those installed, you need to create an npm project by going to an empty folder, running `npm init`, and following its instructions to install Hardhat. Once your project is ready, you should run:

```
npm install --save-dev hardhat
```

To create your Hardhat project, run `npx hardhat` in your project folder. Let's create the sample project and go through these steps to try out a sample task and compile, test and deploy the sample contract.

:::note

The sample project used here comes from the Hardhat Quickstart guide, as well as its instructions.

:::

Creating a project

To create a sample project, run `npx hardhat` in your project folder. You should see the following prompt:

img

Choose the JavaScript project and go through these steps to compile, test and deploy the sample contract.

Checking the contract

The `contracts` folder contains `Lock.sol`, which is a sample contract which consistis of a simple digital lock, where users could only withdraw funds after a given period of time.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.9;

// Import this file to use console.log
import "hardhat/console.sol";

contract Lock {
    uint public unlockTime;
    address payable public owner;

    event Withdrawal(uint amount, uint when);

    constructor(uint _unlockTime) payable {
        require(
            block.timestamp < _unlockTime,
            "Unlock time should be in the future"
        );

        unlockTime = _unlockTime;
        owner = payable(msg.sender);
    }

    function withdraw() public {
        // Uncomment this line to print a log in your terminal
        // console.log("Unlock time is %o and block timestamp is %o", unlockTime, block.timestamp);

        require(block.timestamp >= unlockTime, "You can't withdraw yet");
        require(msg.sender == owner, "You aren't the owner");

        emit Withdrawal(address(this).balance, block.timestamp);

        owner.transfer(address(this).balance);
    }
}
```

Setting up the contract

- Go to `hardhat.config.js`
- Update the `hardhat-config` with matic-network-credentials
- Create `.env` file in the root to store your private key
- Add Polygonscan API key to `.env` file to verify the contract on Polygonscan. You can generate an API key by creating an account

```
require('dotenv').config();
require("@nomiclabs/hardhat-ethers");
require("@nomiclabs/hardhat-etherscan");

module.exports = {
  defaultNetwork: "polygon_mumbai",
  networks: {
    hardhat: {
    },
    polygon_mumbai: {
      url: "https://rpc-mumbai.maticvigil.com",
      accounts: [process.env.PRIVATE_KEY]
    }
  },
  etherscan: {
    apiKey: process.env.POLYGONSCAN_API_KEY
  },
  solidity: {
    version: "0.8.9",
    settings: {
      optimizer: {
        enabled: true,
        runs: 200
      }
    }
```

```
  },
}
```

:::note

Note that the file above requires DOTENV, for managing environment variables and also ethers and etherscan. Make sure to install all those packages.

Find more instructions on how to use DOTENV on this page.

You can deploy on MATIC(Polygon mainnet) if you change polygon_mumbai by MATIC

:::

Compiling the contract

To compile the contract, you first need to install Hardhat Toolbox:

```
npm install --save-dev @nomicfoundation/hardhat-toolbox
```

Then, simply run to compile:

```
npx hardhat compile
```

Testing the Contract

To run tests with Hardhat, you just need to type the following:

```
npx hardhat test
```

And this is an expected output:

img

Deploying on Polygon Network

Run this command in root of the project directory:

```
npx hardhat run scripts/deploy.js --network polygon_mumbai
```

The contract will be deployed on Matic's Mumbai Testnet, and you can check the deployment status here: https://mumbai.polygonscan.com/

Congratulations! You have successfully deployed Greeter Smart Contract. Now you can interact with the Smart Contract.

:::tip Quickly Verify contracts on Polygonscan

Run the following commands to quickly verify your contract on Polygonscan. This makes it easy for anyone to see the source code of your deployed contract. For contracts that have a constructor with a complex argument list, see here.

```
npm install --save-dev @nomiclabs/hardhat-etherscan
npx hardhat verify --network polygon_mumbai 0x4b75233D4FacbAa94264930aC26f9983e50C11AF
```

:::

id: remix title: How to Deploy a Smart Contract Using Remix sidebar_label: Get Started with Remix description: Deploy a Smart Contract with Remix keywords: - docs - matic - polygon - remix - deploy - smart contract image: https://wiki.polygon.technology/img/polygon-logo.png

import useBaseUrl from '@docusaurus/useBaseUrl';

Overview

This tutorial guides you to implement a **Hello World** dApp which echoes a message passed to the contract on to the frontend. You will also be able to change the message using the interactive panel.

:::tip Recommended

We recommend you to follow this tutorial using the online IDE available at Remix IDE. Remix IDE is an easy-to-use platform that does not require any downloads, creating accounts, or logins.

:::

What you will do

- Create a file on Remix
- Upload a pre-built smart contract into the IDE
- Compile the smart contract

- Connect the application to the Polygon Mumbai Testnet via Metamask
- Deploy the smart contract
- Verify the smart contract

Getting started with [Remix IDE]

Remix is a Ethereum-focused IDE: an online platform to develop and deploy smart contracts. To start building a smart contract, click on **New File** and name it `HelloWorld.sol` .

<img src={useBaseUrl("img/remix/new-file.png")} />

Smart Contract

Copy and paste the Smart Contract code provided below into the newly created `HelloWorld.sol` file.

```
// Specifies that the source code is for a version
// of Solidity greater than 0.5.10
pragma solidity ^0.5.10;

// A contract is a collection of functions and data (its state)
// that resides at a specific address on the Ethereum blockchain.
contract HelloWorld {

    // The keyword "public" makes variables accessible from outside a contract
    // and creates a function that other contracts or SDKs can call to access the value
    string public message;

    // A special function only run during the creation of the contract
    constructor(string memory initMessage) public {
        // Takes a string value and stores the value in the memory data storage area,
        // setting `message` to that value
        message = initMessage;
    }

    // A publicly accessible function that takes a string as a parameter
    // and updates `message`
    function update(string memory newMessage) public {
        message = newMessage;
    }
}
```

The first line `pragma solidity ^0.5.10` specifies that the source code is for a Solidity version greater than 0.5.10. Pragmas are common instructions for compilers about how to treat the source code (e.g., pragma once).

A contract in the sense of Solidity is a collection of code (its functions) and data (its state) that resides at a specific address on the Ethereum blockchain. The line `string public message` declares a public state variable called `message` of type `string` . You can think of it as a single slot in a database that you can query and alter by calling functions of the code that manages the database. The keyword `public` automatically generates a function that allows you to access the current value of the state variable from outside of the contract. Without this keyword, other contracts have no way to access the variable.

The constructor is a special function run during the creation of the contract and cannot be called afterward. In this case, it takes a string value `initMessage` , stores the value in the memory data storage area, and sets `message` to that value.

The `update` function is another public function that is similar to the constructor, taking a string as a parameter, and updating the `message` variable.

Compile Smart Contract

- Go to the **Solidity Compiler** tab (below the search button)
- Select compiler version to **0.5.10**
- Now, compile `HelloWorld.sol`
- After successful compilation, it will show a green tick mark on the **Compiler** tab button

Deploying to the Mumbai Testnet

Now, we have to deploy our smart contract on Mumbai, Polygon Network's Testnet. Not only does it cost money (e.g., gas fees) to deploy a smart contract on Polygon Mainnet, but also, the contract is immutable and can't be changed once deployed. Therefore, it's best to first deploy your smart contract to the Testnet first.

To deploy to the Mumbai testnet, we have to connect to the Web3 world which can be accomplished by using any of the services like Metamask, Brave, Portis, etc. We will be using MetaMask in this tutorial. Please follow this guide to set up a MetaMask Account.

- Open Metamask. Click on the network dropdown menu (set to **Ethereum Mainnet** by default) and click on the **Add Network** button. MaticVigil provides a public endpoint and is rate-limited. Therefore, most developers use a free blockchain provider like Alchemy or Quicknode to have a dedicated RPC endpoint with faster throughput AKA **computing units per second (CUPs)**. You need to create a free account to get your private API key from one of these providers to put into the dedicated RPC endpoint URL below.
    - **Network:** Polygon Mumbai Testnet

    - **RPC URL (public endpoint):** https://rpc-mumbai.maticvigil.com.
    - **RPC URL (dedicated endpoint):** https://polygon-mumbai.g.alchemy.com/v2/your-api-key. You'll need a free API key
    - **Chain ID:** 80001
    - **Currency Symbol:** MATIC
    - **Block Explorer URL:** https://mumbai.polygonscan.com/
- Go ahead and click **Save**
- Copy your wallet address from MetaMask by clicking over your account name
- Head over to Faucet and request test MATIC - you will need this to pay for gas on the Polygon network. Select **Mumbai** as the network and **MATIC Token** as the token in the faucet. You can also use this free Mumbai faucet
- Finally, to deploy to Mumbai, refer to the instructions in the "Remix deployment" section below

Deploying to Polygon Mainnet

After you can deploy on the Mumbai Testnet without any errors (i.e., the purpose of the Testnet), let's deploy the Smart Contract on the Polygon Mainnet. Remember this will cost real $ in the form of MATIC tokens.

Similar to the steps above when you deployed to Mumbai, you open your MetaMask wallet. Click on the network dropdown menu (set to **Ethereum Mainnet** by default) and click on the **Add Network** button. Make sure to have your Alchemy API key.

- Open Metamask
    - **Network Name**: Polygon Mainnet
    - **New RPC URL**: https://polygon-mainnet.g.alchemy.com/v2/your-api-key
    - **Chain ID**: 137
    - **Currency Symbol**: MATIC
    - **Block Explorer URL**: https://polygonscan.com/
- Go ahead and click **Save**
- Copy your wallet address from MetaMask by clicking over your account name
- Make sure your MetaMask wallet has MATIC tokens to pay the deployment / gas fees
- Finally, to deploy to Mumbai, refer to the instructions in the "Remix deployment" section below

Remix Deployment

In both Testnet and Mainnet, you do the below to deploy your smart contract using Remix. The below step will use the connect MetaMask API keys you set up in the previous steps.

- Select **Injected Provider** MetaMask in the **Environment** dropdown and your contract

  RemixIDE_Step1

- Accept the **Connect** request received in MetaMask. If the popup doesn't open by default, you can also try manually launching the MetaMask extension
- Once MetaMask is connected to Remix, the `Deploy` transaction would generate another MetaMask popup that requires transaction confirmation. Simply confirm the transaction!

RemixIDE_Step1

Congratulations! You have successfully deployed the **HelloWorld Smart Contract to the **Polygon Mumbai Testnet** You can start interacting with your Smart Contract. Check the deployment status at https://mumbai.polygonscan.com/.

Verifying your Contract

Flatten your Smart Contract

The first and foremost step is to flatten the solidity smart contract into a single file. In order to do that, install truffle-flattener or sol-merger.

After installation, flatten the contract using below command (we have demonstrated using sol-merger).

```
sol-merger \"./contracts/*.sol\" ./build
```

Verifying on Polygonscan

- Navigate to your contract's Polygonscan page and click on **Verify and Publish**.

  <img src={useBaseUrl("img/verification/verify-publish.png")} />

- Select **Solidity (Single File)** in compiler type
- Select appropriate compiler version
- Choose the license type of your contract

In the next section, paste your flattened samrt contract here. If you had enabled optimization, then adjust the `optimization` section accordingly.

Constructor arguments should have been filled in automatically. If not, they can be retrieved from the trailing bytes of the deployment transaction (example: `000000000000000000000000a6fa4fb5f76172d178d61b04b0ecd319c5d1c0aa`).

---

id: replit title: How to Deploy a Smart Contract Using Replit sidebar_label: Get Started with Replit description: Deploy Smart Contracts using ReplitIDE on Polygon keywords: - docs - matic - replit - deploy - smart contract - polygon - IDE image: https://wiki.polygon.technology/img/polygon-logo.png

import useBaseUrl from '@docusaurus/useBaseUrl';

Overview

Replit is a coding platform that allows you to write code and host apps. Replit supports Solidity programming language so it provides all of the features and functionality for Web3 developers to create and deploy smart contracts.

This article guides you to build and deploy a solidity smart contract on Polygon using Replit IDE and Replit Solidity development template (Solidity starter beta).

What you will do

- Create a Replit account
- Create a Repl environment
- Deploy a sample project on the Polygon Mumbai network
- Verify the contract
- Publish your project to a personal Replit profile.

:::tip

For additional examples about Solidity with Replit, you can read the article **Get started with Replit** or check **Replit Solidity documentation and Escrow contract tutorial**. :::

Prerequisites

You do not need any local environment setup to deploy your solidity smart contract on Polygon using Replit.

You need a browser-based Web3 wallet to interact with the Polygon Mumbai Testnet and deployed contracts. If you are already using MetaMask, it is recommended to create a new account for testing with Replit. You can do this from the account menu, which appears when you click on the account avatar in the top right corner of MetaMask interface.

You must set up all of the following prerequisites to be able to deploy your solidity smart contract on Polygon:

1. Create a Replit account
2. Download MetaMask wallet
3. Configure Polygon on MetaMask
4. Get testnet tokens

Working with a Repl

Every Repl that you create is a fully functional development and production environment. Follow the steps to create a solidity starter Replit:

1. Log in or create an account. After creating your Replit account, your home screen will include a dashboard where you can view, create projects, and manage your account.

img

2. Once logged in, create a Solidity starter repl, Select **+ Create Repl** from the left panel or **+** in the top right corner of the screen.

img

3. Select the **Solidity starter (beta)** template and give your project a title.

4. Click on **+ Create Repl** to create your project.

:::note

The Solidity starter repl comes with a browser-friendly interface, built using the **Web3 Ethereum JavaScript API**, which you can use to deploy and interact with our contracts. We will deploy to Replit's testnet, a custom version of the Ethereum blockchain managed by Replit and optimised for testing.

:::

Deploy on Polygon

Make sure that you have followed the list of **Prerequisites** above so that you are ready to deploy and interact with your smart contract.

1. Click on **Run** (at the Top) to install all relevant packages and start up the contract deployment UI.

2. Connect your MetaMask wallet to the web interface and switch to the Mumbai Testnet.

img

3. Click on **Connect wallet**, select your account, then choose **Connect**.

img

4. From the dropdown list, select the contract that you want to deploy. Click on **Deploy**.

5. You will receive a MetaMask popup window asking for your confirmation. Approve the transaction from your wallet to deploy your contract.

Verifying and Testing your Contract

When the contract is deployed, navigate to Polyganscan to search for your account, view your deployed contract, and copy your account address.

Once your contract has been deployed, it will show up as expandable boxes below the dropdown box. Expand it and take a look at all the different functions available. You can now interact with your contract using the provided user interface or from a sharable URL shown on the interface.

Publish to Replit

Replit allows you to publish your projects to a personal profile. After publishing, projects will show up on your spotlight page for others to explore, interact with, clone, and collaborate.

Follow the below steps to publish your projects to Replit:

1. Select the project title at the top of the screen.
2. Complete your project name and description and click **Publish**.

---

id: truffle title: How to Deploy a Smart Contract Using Truffle sidebar_label: Get Started with Truffle description: Use Truffle to deploy a Smart Contract on Polygon keywords: - docs - matic - polygon - smart - contract - truffle - deploy - polygonscan image: https://wiki.polygon.technology/img/polygon-logo.png

Overview

Truffle is a blockchain development environment, which you can use to create and test smart contracts by leveraging the Ethereum Virtual Machine. This guide aims at teaching how to create a smart contract using Truffle and deploying it on the EVM-compatible Polygon Network.

:::note

This tutorial is an adapted version of the Truffle quickstart guide article.

:::

What you will do

- Install and set up Truffle
- Deploy contract on Polygon Network
- Check the deployment status on Polygonscan

Prerequisites

There are a few technical requirements before we start. Please install the following:

- Node.js v8+ LTS and npm (packaged with Node)
- Git

Once we have those installed, we only need one command to install Truffle:

```
npm install -g truffle
```

To verify that Truffle is installed properly, type `truffle version` on a terminal. If you see an error, make sure that the npm modules are added to your path.

Creating a Project

MetaCoin Project

We will use one of Truffle's boilerplates which you can find on their Truffle Boxes page. MetaCoin box creates a token that can be transferred between accounts.

1. Start by creating a new directory for this Truffle project:

```
mkdir MetaCoin
cd MetaCoin
```

2. Download the MetaCoin box:

```
truffle unbox metacoin
```

With that last step, you have created a Truffle project cointaining folders with contracts, deployment, testing, and configuration files.

This is the smart contract data from the `metacoin.sol` file:

```
// SPDX-License-Identifier: MIT
// Tells the Solidity compiler to compile only from v0.8.13 to v0.9.0
pragma solidity ^0.8.13;

import "./ConvertLib.sol";

// This is just a simple example of a coin-like contract.
// It is not ERC20 compatible and cannot be expected to talk to other
// coin/token contracts.

contract MetaCoin {
    mapping (address => uint) balances;

    event Transfer(address indexed _from, address indexed _to, uint256 _value);

    constructor() {
        balances[tx.origin] = 10000;
    }

    function sendCoin(address receiver, uint amount) public returns(bool sufficient) {
        if (balances[msg.sender] < amount) return false;
        balances[msg.sender] -= amount;
        balances[receiver] += amount;
        emit Transfer(msg.sender, receiver, amount);
        return true;
    }

    function getBalanceInEth(address addr) public view returns(uint){
        return ConvertLib.convert(getBalance(addr),2);
    }

    function getBalance(address addr) public view returns(uint) {
        return balances[addr];
    }
}
```

:::note

Notice that ConvertLib is being imported just after the `pragma` statement. In this project, there are actually two smart contracts that will be deployed at the end: one is Metacoin, contatining all the send and balance logic; the other is ConvertLib, a library used to convert values.

:::

Testing the Contract

You can run Solidity and Javascript tests.

1. In a terminal, run the Solidity test:

```
truffle test ./test/TestMetaCoin.sol
```

You should see the following output:

img

2. Run the JavaScript test:

```
truffle test ./test/metacoin.js
```

You should see the following output:

img

Compiling the Contract

Compile the smart contract using the following command:

```
truffle compile
```

You will see the following output:

img

Configuring the Smart Contract

Before actually depolying the contract, you need to set up the `truffle-config.js` file, inserting network and compilers data.

Go to `truffle-config.js` and update the file with Polygon Mumbai network details.

```
const HDWalletProvider = require('@truffle/hdwallet-provider');
const fs = require('fs');
const mnemonic = fs.readFileSync(".secret").toString().trim();

module.exports = {
  networks: {
    development: {
      host: "127.0.0.1",     // Localhost (default: none)
      port: 8545,            // Standard Ethereum port (default: none)
      network_id: "*",       // Any network (default: none)
    },
    matic: {
      provider: () => new HDWalletProvider(mnemonic, `https://rpc-mumbai.maticvigil.com`),
      network_id: 80001,
      confirmations: 2,
      timeoutBlocks: 200,
      skipDryRun: true
    },
  },

  // Set default mocha options here, use special reporters etc.
  mocha: {
    // timeout: 100000
  },

  // Configure your compilers
  compilers: {
    solc: {
        version: "0.8.13",
    }
  }
}
```

Note that it requires mnemonic to be passed in for `maticProvider`. This is the seed phrase (or private key) for the account you would like to deploy from. Create a new `.secret` file in the root directory and enter your 12-word mnemonic seed phrase to get started. To get the seed words from MetaMask wallet, you can go to MetaMask settings, then from the menu, choose **Security and Privacy** where you will see a button that says **reveal seed words**.

Deploying on Polygon Network

Add MATIC to your wallet using Polygon Faucet. Next, run this command in the root folder of the project directory:

```
truffle compile
truffle deploy --network matic
```

img

:::note

Remember your `address`, `transaction_hash` and other details provided would differ. Above is just to provide an idea of the structure.

:::

Congratulations! You have successfully deployed a Smart Contract using Truffle. Now you can interact with the contract and also check its deployment status on Polygonscan.

id: wallet-bridge-faq title: Wallet <> Bridge FAQ description: Build your next blockchain app on Polygon. keywords: - docs - matic - polygon - wiki - wallet image: https://wiki.polygon.technology/img/polygon-logo.png

import useBaseUrl from '@docusaurus/useBaseUrl';

Where can I use the Polygon Web Wallet?

Here is the Polygon Wallet Suite URL: https://wallet.polygon.technology/ The Polygon Wallet Suite is a collection of Web3 applications provided by Polygon. It consists of the Polygon Wallet (a decentralized wallet), Polygon Bridge (an L1-L2 bridge), Polygon Staking (an environment for staking and delegating MATIC tokens) and Polygon Safe Bridge (a multisig bridge).

Which wallets are currently supported?

Metamask, Coinbase, Bitski Wallet, Venly and WalletConnect are the currently supported wallets.

What can I do with my Polygon wallet?

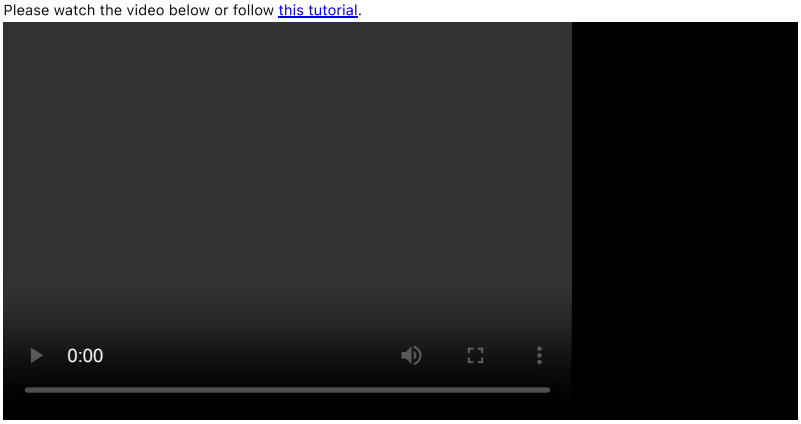
- Send funds to any account on Polygon.
- Deposit funds from Ethereum to Polygon (using the bridge).
- Withdraw funds back to Ethereum from Polygon (also using the bridge).

My MetaMask wallet is not connecting with Polygon wallet

There are many reasons why this might be happening. We suggest that you **try another time**, **use another browser** or, if any of these doesn't help, **contact our support team**.

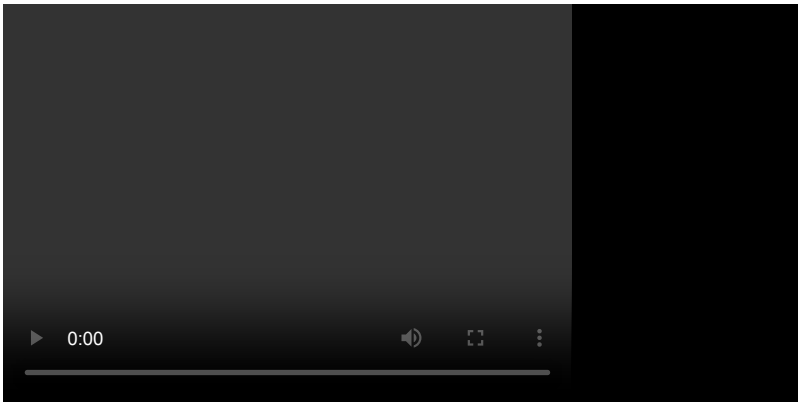
How can I deposit Funds from Ethereum to Polygon using Polygon Wallet Suite.

Please watch the video below or follow this tutorial.



How can I withdraw funds from Polygon to Ethereum via PoS Bridge using Polygon Wallet Suite?

Please watch the video below or follow this tutorial.



How to add a new or custom token to Polygon Wallet Token list?

Please follow this tutorial.

How do I find the token contract?

The token contract address will be required when you are trying to add a new or custom token. You can search for the token by its name on either Coingecko or CoinMarketCap where you will be able to see its address on the Ethereum chain (for ERC20 tokens) and other supported blockchains like Polygon. The token address on other chains might not be updated but you can surely use the root address for all purposes.

I have deposited my funds but I don't see it on Metamask. What do I do?

You need to manually add the custom token address to Metamask.

Open MetaMask and scroll down to click on **Import tokens**.

Then, add the relevant contract address, symbol, and decimal precision. Contract addresses (PoS-WETH in this case) can be found on this link: https://docs.polygon.technology/docs/operate/mapped-tokens/. You will need to add the child token address to view balances on Polygon Mainnet. The decimal of precision is 18 for WETH (for most tokens, the decimal of precision is 18).

How can I add Polygon Mainnet on Metamask?

Check this tutorial.

My token is not visible in the list. Who should I contact?

Reach out to the Polygon team on Discord or Telegram and get your token listed. Before that, ensure your token is mapped. If it is not mapped, please raise a request at https://mapper.polygon.technology/.

Can I cancel my transaction after the checkpoint arrived?

Once the withdrawal transaction is initiated on the Polygon Mainnet, then unfortunately it can't be canceled or reverted. In withdrawal transactions, tokens are burned from the Polygon Mainnet and minted on the Ethereum Mainnet. Therefore, tokens once burned from the Polygon chain can't be reverted back to your wallet.

The gas fee is too high, can I cancel my transaction?

Unfortunately, we can not cancel the withdrawal transaction once tokens are burned from the Polygon Mainnet. In other words, it is impossible to cancel a transaction once it is initiated. The gas fee is not controlled by the Polygon. It is totally dependent on the network congestion and the number of transactions in a particular block on the Ethereum Mainnet. If you think you can not afford the current gas fee, you can wait and try to proceed with your transaction later when the gas fee is on the lower side. You can also monitor the gas fee on the Ethereum Mainnet from here: https://etherscan.io/gastracker

Can I send my tokens from Polygon to any other wallet/exchange ?

You cannot directly send tokens from Polygon UI to exchanges/wallets. You have to first withdraw from Polygon to Ethereum and then send it to your exchange address (unless your exchange/wallet explicitly supports the network).

I made the mistake of sending funds to an exchange/wallet directly. Can you help?

Unfortunately, we can not assist in such cases. Please don't send funds directly to exchanges that support only Ethereum, you have to first withdraw from Polygon to Ethereum and then send it to your exchange address.

I made a transfer to the wrong address. How do I retrieve the funds?

Unfortunately, nothing can be done. Only the owner of the private keys to that particular address can move those assets. It's always advisable to confirm if the address you are sending tokens to is the right one.

My transaction has been pending for too long, what can I do?

The transaction might be dropped due to the following reasons:

1. Setting up a low gas price while submitting the transaction.
2. A sudden surge in the gas price due to congestion on the Ethereum Mainnet.
3. The transaction is canceled by you from your wallet or replaced with a new transaction.

You can proceed with the dropped transactions in the following ways:

1. If your transaction is stuck for more than an hour, a **Try Again** button will be shown. You can click on the **Try Again** button to complete the same transaction. You can refer to this video for more information on how to use the **Try Again** feature.
2. Please check your MetaMask wallet as well because sometimes transactions might be dropped due to queued-up transactions in the Metamask. In that case, clear the queued-up transactions or re-install the MetaMask in the same browser.
3. You can install the MetaMask in an alternate browser and then try to complete the transaction using Polygon Wallet Suite.
4. You can also use this link to complete the pending withdrawal transaction. Paste the transaction hash in the search option and click the **Confirm Exit** button to complete the transaction.

What do I do if the deposit is confirmed but the balance is not getting updated?

It takes 22-30 minutes for the deposit transaction to complete. Please wait for some time and click on **Refresh Balance**.

What should I do if the checkpoint is not happening?

Checkpoints sometimes take more than 45 minutes to 1 hour based on network congestion on Ethereum, we suggest waiting for a while before raising a ticket.

My transaction is stuck.

We have listed some common errors that users might face. You can find the solution below the image of the error. In case you're shown a different error, please raise a support ticket for our team to troubleshoot.

Common Errors

<img src={useBaseUrl("img/wallet-bridge/checkpoint-rpc-error.png")} width="357" height="600"/>

The current RPC error you're facing might be due to an RPC overload.

Please try changing your RPC and proceed with the transaction. You may follow this link here for more information.

<img src={useBaseUrl("img/wallet-bridge/checkpoint-stumbled-error.png")} width="357" height="600"/>

This is usually an off-and-on error that gets resolved automatically. In case you are still receiving the same error while reinitiating the step, do raise a support ticket with all the relevant information to troubleshoot this further.

I'm shown an insufficient balance error.

Withdrawals and deposits on the Polygon network are cheap. What is to be understood is that the insufficient balance error can be cleared by getting some ETH balance on the Ethereum Mainnet. That generally clears out the problem of an insufficient balance. If this is a transaction on the Polygon Mainnet, we'll require that you have a sufficient amount of MATIC tokens.

My transactions are not visible on the explorer. What should I do?

This is probably an indexing issue with Polygonscan. Please contact the Support Team for more clarifications.

I initiated a deposit on Ethereum but it still shows as pending. What should I do?

Your supplied gas is probably too low. You should wait a while and redo the transaction if it doesn't get mined. In case of additional help, please reach out to the support team with your wallet address, transaction hashes (if any) and relevant screenshots.

I'm not getting a transaction hash and my deposits aren't going through? What is happening?

You probably have prior pending transactions, please cancel or speed them up first. Transactions in Ethereum can only happen one after the other.

It shows Polygon does not charge any amount for a withdrawal but we are to pay during the transaction.

On the PoS bridge, the withdrawal transaction happens over two steps: Token burning on the Polygon network and proof submission on the Ethereum network. Token burning that happens on the Polygon Mainnet will be a minimal cost. The remaining steps that happen on the Ethereum Mainnet will have to be paid in ETH depending on the current gas price which can be verified here.

I was trying to make a deposit but the transaction stopped at the Approve step.

If the transaction is still at the **Approve** step, it is not yet complete. To fulfill it, you need to pay the gas fee and then it should go through.

Polygon wallet shows 'User denied transaction signature' error message.

This usually happens because the user canceled or refused to sign a transaction via MetaMask. When prompted by the MetaMask wallet, proceed with signing the transaction by clicking on **Approve** and not on **Cancel**.

The transaction is successful but it shows pending.

If your transaction is completed and you received your funds but still the transaction shows pending on the UI, you can raise a support ticket by sending relevant details and screenshots.

What is the list of Supported Exchanges on Polygon?

The MATIC coin can be traded in many exchanges. However, it's always important to do your own research when you are choosing one to trade. It is not unusual that some exchanges keep making changes to their current available tokens and also have maintenance periods.

You might visit Coinmarketcap for a list of exchanges where you might find MATIC.

Does Polygon support hardware wallets?

Yes, we support the following hardware wallets:

1. Trezor
2. Ledger

Users can connect their Hardware wallet option on MetaMask and proceed with their transaction. Here is the link to connect the hardware wallet on Metamask: https://metamask.zendesk.com/hc/en-us/articles/4408552261275

Why isn't the MATIC token supported on PoS?

MATIC is the native token of Polygon and it has a contract address - 0x0000000000000000000000000000000000001010 on the Polygon chain. It is also used to pay for gas. Mapping the MATIC token on the PoS bridge will lead to MATIC having an additional contract address on the Polygon chain. This will collide with the existing contract address as this new token address can not be used to pay for gas and will have to remain as a normal ERC20 token on the Polygon chain.

How do I map tokens?

Please refer to [this tutorial] (/docs/pos/design/bridge/ethereum-polygon/submit-mapping-request) or you could go straight to the Token Mapper.

What do I do if the transaction is taking too long or if the gas price is too high?

Transaction time and gas price varies based on network congestion and it is also determined by supply and demand between the network's miners.

What you could do:

- Be patient.

- Increase the gas fee if it is too slow.
- Check the fees before sending transactions. Here is a link for Etherscan's gas tracker: https://etherscan.io/gastracker

What you shouldn't do:

- Please do not set the gas limit low or your transaction might fail.
- Do not attempt to cancel the transaction. Check the fees beforehand.

Can I change the gas limit or the gas price?

The gas limit is estimated and set by the application according to certain requirements of the function being called in the contract. This should not be edited. Only the gas price can be changed in order to increase or decrease the transaction fees.

How to speed up the transactions?

You can do so by increasing the gas fees. Here's a link explaining how to do it on Metamask: https://metamask.zendesk.com/hc/en-us/articles/360015489251-How-to-Speed-Up-or-Cancel-a-Pending-Transaction.

How much MATIC token is enough for the gas fee?

Users need to have a minimum of 0.01 MATIC in the Polygon mainnet.

Where do I raise a support ticket?

If you need help from our specialists, please send us a message at https://support.polygon.technology/support/home.

How do I bridge assets across chains?

Polygon offers a bridge to move assets from Ethereum to Polygon and vice versa. You can learn more about it on the Bridges section of this wiki.

However, if you are using any external service that is not owned by Polygon, we advise you to reach out to their customer service to request tutorials and instructions. It is also important to do your own research when you are using web3 services.

I have a token withdrawal issue with OpenSea or any other application which uses polygon bridge.

If you have an issue with your withdrawal transaction being stuck, Polygon offers the withdraw bridge with https://withdraw.polygon.technology to help get you off the ground if you have your burn hash. With this tool, you're quickly onboarded and the issue will be resolved. Other questions concerning your transaction with OpenSea and other dApps will have to be handled by the application team.

I have been scammed. How will I retrieve my tokens?

Unfortunately, there is no recovery process for lost coins. We ask that before you make a transaction, you go on to check and double-check before starting and completing it. Please note that the Polygon network and our official handles do not engage in any giveaway posts or token doubling and we will never approach you on behalf of the organization. Please disregard all attempts as they're most likely scams. All our communications are through our official handles.

There are some unauthorized transactions in my wallet. Is my wallet hacked?

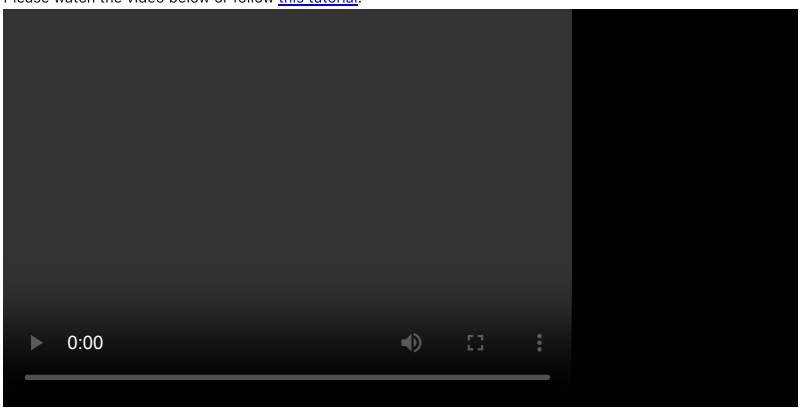
Unfortunately, the network cannot revert unwanted transactions. It is always important to be careful with your private keys and **never share them with anyone**. If you still have some remaining funds, transfer them immediately to a new wallet.

Ethereum has Goerli as its test network. Does Polygon Network have a test network too?

As the Ethereum Network has Goerli as its test network, the Polygon Mainnet has Mumbai. All transactions on this test network will be indexed on the Mumbai Explorer.

How can I swap my tokens for other tokens?

Please watch the video below or follow this tutorial.



The Token Swap is too slow.

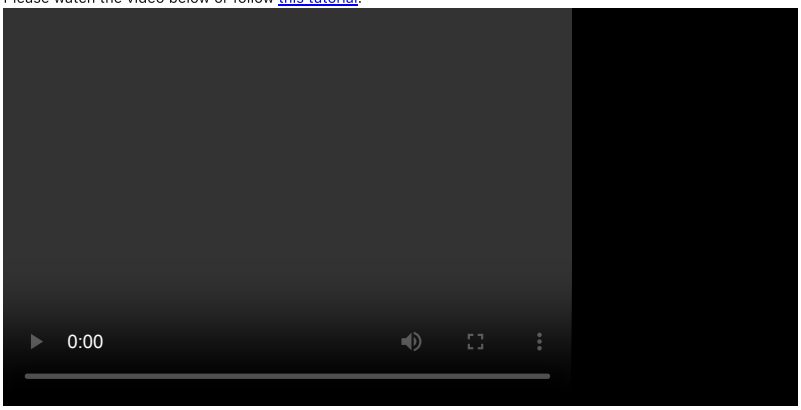
If you are trying to swap tokens and it is taking too long, you could try the same transaction on a different browser. If that doesn't work and you're facing an error, please send a screenshot to our Support team.

Which tokens are charged as the gas fees for token swap?

Only MATIC.

How can I swap my token for gas?

Please watch the video below or follow this tutorial.



Which tokens can be used to swap for gas?

Only these Tokens are supported for 'Swap for Gas': ETH, USDC, USDT, DAI, AAVE, LINK, WBTC, UNI, GHST, TEL, EMON, and COMBO.

How to get ETH tokens?

To acquire ETH tokens, you can either trade them for another token or fiat money on an exchange, buy them on an on-ramp (or on Metamask) or even swap other tokens for ETH using Polygon's token swap feature.

How can I get MATIC tokens to pay for gas fees?

We provide a Gas Swap service that will help you with that. You choose an amount of MATIC you need to complete your transaction and you can swap it for other tokens such as Ether or USDT. It's worth noting that this is a **gas-less transaction**.

Where can I get MATIC tokens directly?

So MATIC tokens can be bought from any centralized (Binance, Coinbase, et.al) or Decentralized (Uniswap, QuickSwap) exchange. You may also research and try some on-ramps like Transak, and Ramp. The purpose for your purchase of MATIC coins should also determine where you'll buy them from and the network. It is advisable to have MATIC on the Ethereum mainnet if your intention is either staking or delegation. If your intent is a transaction on the Polygon Mainnet, you should hold and transact with MATIC on the Polygon Mainnet.

---

id: matic-faucet title: MATIC Faucet sidebar_label: Test MATIC Faucet description: Request MATIC tokens on the Mumbai Testnet using Polygon, Alchemy, and Quicknode. keywords: - docs - polygon - wiki - tokens - faucet - testnet - matic - alchemy - quicknode image: https://wiki.polygon.technology/img/polygon-logo.png

import useBaseUrl from '@docusaurus/useBaseUrl';

Polygon Faucet is a tool provided by **Polygon** and its ecosystem partners such as **Alchemy** and **Quicknode** that **allows you to obtain free MATIC tokens on the Mumbai Testnet** and get started with the Polygon network. **These test tokens enable you to work with Polygon features without having to spend actual MATIC tokens** on the Mainnet.

:::info

Tokens on Testnet networks like Mumbai and Goerli have no value as they are only used for testing purposes.

:::

We have listed three faucet tools you can use to receive **MATIC Testnet tokens**, along with their usage guides, in the sections below.

- Polygon MATIC Faucet
- Alchemy MATIC Faucet
- QuickNode Multi-chain Faucet

Polygon MATIC Faucet

- Navigate to **faucet.polygon.technology** <img src={useBaseUrl("img/tools/faucet.png")} />
- Select one of the blockchain testnet networks, where:

    - **Mumbai** - Polygon Testnet
    - **Goerli** - Ethereum Testnet
- Select the type of Testnet token that you want to receive, where:
    - **MATIC Token** - Testnet token of Polygon network
    - **Test ERC20** - Standard testnet token on the Ethereum network
    - **Test ERC1155** - Standard testnet token used for NFTs
    - **LINK** - ERC677 testnet token that inherits functionality from the ERC20
- Enter your wallet address (you can copy it from your MetaMask or Polygon wallet)
- Click on the **Submit** button to send your token request
- Click **Confirm** to finalize the transaction

  <img src={useBaseUrl("img/tools/confirm-transaction.png")} />

:::caution

If you don't have enough MATIC Testnet tokens in your account to pay for the gas fees, the transaction may fail. If you require testnet tokens in bulk, please fill **this form**.

:::

- After confirmation, you will receive the requested Testnet tokens within 1 to 2 minutes.

  <img src={useBaseUrl("img/tools/success.png")} />

Alchemy MATIC Faucet

:::tip GET 5 MATIC

If you sign up or log in with an Alchemy account, you get 5 times more MATIC/day (**up to 5 MATIC**).

:::

- You can request MATIC tokens from **Alchemy's Mumbai Faucet**. In order to receive Testnet tokens, head over to the **Mumbai Faucet website** by Alchemy, enter your **wallet address** and hit the **Send Me MATIC** button.

img

- You can check the transaction hash as soon as you finish requesting your MATIC and also view the transaction details on Polygonscan.

QuickNode Multi-chain Faucet

In order to receive MATIC Mumbai tokens using QuickNode Fauct,

- Head over to QuickNode Multi-chain Faucet. Connect your wallet (MetaMask or Phantom) by clicking on **Connect Wallet**. Select Polygon and Mumbai Testnet from the dropdown menus, and then click on **Continue**.

  img

:::tip Get 1 MATIC

If you **tweet** the drip faucet URL along with your wallet address and submit the tweet URL, you get 10X more MATIC Testnet tokens (i.e. 1 MATIC).

:::

- To receive 0.1 Testnet MATIC, click on **No thanks, just send me 0.1 MATIC**.

  img

- You can also view your transaction details on Polygonscan by clicking on the **TxHash**.

---

id: polygon-gas-station title: Polygon Gas Station sidebar_label: Polygon Gas Station description: Polygon Gas Station aims to help dApp developers with gas price recommendations. keywords: - docs - matic - polygon - wiki - gas recommendation - dapps image: https://wiki.polygon.technology/img/polygon-logo.png

import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem';

Polygon Gas Station aims to help dApp developers with gas price recommendations, so that they can use it before sending transaction off to the **Polygon** network.

We received a lot of requests from dApp developers for building a gas price recommendation service. So we took some inspiration from **Eth Gas Station** and built one.

Polygon Gas Station has been deployed both on the Polygon Mainnet & Mumbai Testnet. It queries the RPC for `eth_feeHistory` and fetches the 10th, 25th, and 50th percentiles of priority fees for transactions in each of the last 15 blocks. The average value of the 10th, 25th, and 50th percentiles become the `safeLow`, `standard`, and fast fee predictions.

Usage

<Tabs defaultValue="mainnet" values={[{ label: 'Polygon Mainnet', value: 'mainnet', }, { label: 'Mumbai Testnet', value: 'mumbai', },] }>

For getting gas price recommendation from this oracle (Polygon Mumbai), send **GET** request to https://gasstation-testnet.polygon.technology/v2

cURL

```
curl https://gasstation-testnet.polygon.technology/v2
```

JavaScript

```
fetch('https://gasstation-testnet.polygon.technology/v2')
  .then(response => response.json())
  .then(json => console.log(json))
```

Python

```
import requests
requests.get('https://gasstation-testnet.polygon.technology/v2').json()
```

For getting gas price recommendation from this oracle (Polygon Mainnet), send **GET** request to the Polygon Gas Station V2 to get the gas fee estimates. Polygon Gas Station V2 Endpoint: https://gasstation.polygon.technology/v2

cURL

```
curl https://gasstation.polygon.technology/v2
```

JavaScript

```
fetch('https://gasstation.polygon.technology/v2')
  .then(response => response.json())
  .then(json => console.log(json))
```

Python

```
import requests
requests.get('https://gasstation.polygon.technology/v2').json()
```

Interpretation

An example JSON response will look like this:

```
{
  "safeLow": {
    "maxPriorityFee":30.7611840636,
    "maxFee":30.7611840796
    },
  "standard": {
    "maxPriorityFee":32.146027800733336,
    "maxFee":32.14602781673334
    },
  "fast": {
    "maxPriorityFee":33.284344224133335,
    "maxFee":33.284344240133336
    },
  "estimatedBaseFee":1.6e-8,
  "blockTime":6,
  "blockNumber":24962816
}
```

- {'safelow', 'standard', 'fast', 'estimatedBaseFee'} are gas prices in GWei. You can use these prices before sending transaction off to Polygon, depending upon your needs.
- `blockNumber` tells what was latest block mined when recommendation was made.
- `blockTime`, in second, gives average block time of the network.

---

id: abi-manager title: ABIManager keywords: - 'abi manager, api type, read, write, polygon' description: "Matic.js's internal ABI manager."

`matic.js` internally uses `ABIManager` for handling ABI management, configuration for you. All of the ABI and config are taken from static repo.

Change ABI

Sometimes you are required to change the ABI, particularly when you are developing a contract. You can do so by using `ABIManager`.

Syntax

```
import { ABIManager } from '@maticnetwork/maticjs'


const manager = new ABIManager(<network name>,<version>);
await manager.init();

// set abi

manager.setABI(<contract name>,<bridge type>, <abi value>);

// get abi

manager.getABI(<contract name>,<bridge type>);
```

The network name, contract name, bridge name etc can be taken from our official static repo.

Example

```
import { ABIManager } from '@maticnetwork/maticjs'


const manager = new ABIManager('testnet','mumbai');
await manager.init();

// set abi

manager.setABI('ERC20PredicateProxy','pos', 'abi value');

// get abi

manager.getABI('ERC20PredicateProxy','pos');
```

---

id: exit-util title: ExitUtil keywords: - 'exit util, api type, read, write, polygon' description: 'Generate proof with the ExitUtil class.'

`matic.js` internally use `ExitUtil` for generating proof. It is a class which has different methods for helping with exit utilities.

buildPayloadForExit

It exposes `buildPayloadForExit` method which can be used to generate proof.

```
import { ExitUtil, RootChain, use, Web3SideChainClient } from "@maticnetwork/maticjs";
import { Web3ClientPlugin } from "@maticnetwork/maticjs-web3";
import HDWalletProvider from "@truffle/hdwallet-provider";
import { from, privateKey, RPC } from "./config";
use(Web3ClientPlugin);


const client = new Web3SideChainClient<any>();
// initiate client
await client.init({
    // log: true,
    network: 'testnet',
    version: 'mumbai',
    parent: {
        provider: new HDWalletProvider(privateKey, RPC.parent),
        defaultConfig: {
            from
        }
    },
    child: {
        provider: new HDWalletProvider(privateKey, RPC.child),
        defaultConfig: {
            from
        }
```

```
    }
});

// create root chain instance
const rootChain = new RootChain(client, <root chain address>);

// create exitUtil Instance
const exitUtil = new ExitUtil(client, rootChain);

// generate proof
const proof = await exitUtil.buildPayloadForExit(
    <burn tx hash>,
    <log event signature>,
    <isFast>
)
```

Generating proof using bridge client

Every bridge client including **POSClient**, **PlasmaClient** exposes `exitUtil` property.

```
import { POSClient,use } from "@maticnetwork/maticjs"
import { Web3ClientPlugin } from '@maticnetwork/maticjs-web3'
import HDWalletProvider from "@truffle/hdwallet-provider"

// install web3 plugin
use(Web3ClientPlugin);

const posClient = new POSClient();
await posClient.init({
    network: 'testnet',
    version: 'mumbai',
    parent: {
      provider: new HDWalletProvider(privateKey, mainRPC),
      defaultConfig: {
        from : fromAddress
      }
    },
    child: {
      provider: new HDWalletProvider(privateKey, childRPC),
      defaultConfig: {
        from : fromAddress
      }
    }
});

const proof = await posClient.exitUtil.buildPayloadForExit(
    <burn tx hash>,
    <log event signature>,
    <isFast>
)
```

---

id: plugin title: Plugin keywords: - 'plugin, api type, read, write, polygon' description: 'Use plugin to inject code into Matic.js.'

Using plugin you can inject your code into `matic.js`. It can be used to write common set of generic codes which can be provided to anyone using a package.

:::info Plugin makes the `matic.js` light weight as it implements only important logical part. :::

In fact, the web3 library is supported using plugin which allows us to use our favourite library.

Plugin development

Plugin is a class which implements `IPlugin`.

```
import { IPlugin } from "@maticnetwork/maticjs";

export class MyPlugin implements IPlugin {

    // variable matic is - default export of matic.js
    setup(matic) {

        // get web3client
        const web3Client = matic.Web3Client ;
    }
}
```

As you can see - you just need to implement a `setup` method which will be called with default export of `matic.js`.

Use Plugin

`matic.js` expose `use` method for using a plugin.

```
import { use } from '@maticnetwork/maticjs'

use(MyPlugin)
```

You can use multiple plugins and they will be called in the same order as they are declared.

Some plugin repos are -

- Matic web3.js
- Matic ethers
- ExPortal.js

---

id: api-architecture title: API Architecture keywords: - api architecture - api type - read - write - polygon description: Read and Write APIs plus transaction settings.

The library follows common api architecture throughout and the APIs are divided into two types -

1. Read API
2. Write API

Read API

Read APIs does not publish anything on blockchain, so it does not consume any gas. Example of read APIs are - `getBalance`, `isWithdrawExited` etc.

Let's see an example of read API -

```
const erc20 = posClient.erc20('<token address>');
const balance = await erc20.getBalance('<user address>')
```

read APIs are very simple and returns result directly.

Write API

Write APIs publish some data on the blockchain, so it consumes gas. Example of write APIs are - `approve`, `deposit` etc.

When you are calling a write API - you need two data from the result.

1. TransactionHash
2. TransactionReceipt

Let's see an example of write API and get the transactionhash and receipt -

```
const erc20 = posClient.erc20('<token address>');

// send the transaction
const result = await erc20.approve(10);

// get transaction hash

const txHash = await result.getTransactionHash();

// get receipt

const receipt = await result.getReceipt();
```

Transaction option

There are some configurable options that are available for all API's. These configurations can be passed in parameters.

Available configurations are -

- from?: string | number - The address transactions should be made from.
- to?: string - The address transactions should be made to.
- value?: number | string | BN - The value transferred for the transaction in wei.
- gasLimit?: number | string - The maximum gas provided for a transaction (gas limit).
- gasPrice?: number | string | BN - The gas price in wei to use for transactions.
- data?: string - The byte code of the contract.
- nonce?: number;
- chainId?: number;
- chain?: string;
- hardfork?: string;
- returnTransaction?: boolean - making it true will return the transaction object which can be used to send transaction manually.

Let's see an example by configuring the gasPrice

```
const erc20RootToken = posClient.erc20(<root token address>,true);

// approve 100 amount
const approveResult = await erc20Token.approve(100, {
    gasPrice: '4000000000',
});

const txHash = await approveResult.getTransactionHash();

const txReceipt = await approveResult.getReceipt();
```

---

id: fx-portal title: FxPortal keywords: - fx-portal - client - contract - polygon - sdk description: Polygon state-sync mechanism with no mapping.

FxPortal Bridge

FxPortal Bridge is similar to POS Bridge with a feature of using ERC standardized tokens without mapping.

For more info about FxPortal, please read - [FxPortal Bridge doc]

FxPortal bridge functionality is available in [seperate repository] as a plugin for maticjs. The package provides different API's for interacting with bridge.

Here is repository link - [https://github.com/fx-portal/maticjs-fxportal]

id: get-started title: Get Started keywords: - maticjs - introduction - contract - polygon - sdk description: Get started with Matic.js

The `@matic.js` is a javascript library which helps in interacting with the various components of Matic Network.

In this Get Started tutorial - we will learn about how we can setup and interact with the POS bridge.

Installation

Install the maticjs package via npm:

```
npm install @maticnetwork/maticjs
```

Install the web3js plugin

```
npm install @maticnetwork/maticjs-web3
```

Setup

```
import { use } from '@maticnetwork/maticjs'
import { Web3ClientPlugin } from '@maticnetwork/maticjs-web3'

// install web3 plugin
use(Web3ClientPlugin)
```

In the above code we are initiating maticjs with `web3js` but you can also similarly initiate with [ethers].

POS client

`POSClient` helps us to interact with POS Bridge.

```
import { POSClient,use } from "@maticnetwork/maticjs"
import { Web3ClientPlugin } from '@maticnetwork/maticjs-web3'
import HDWalletProvider from "@truffle/hdwallet-provider"

// install web3 plugin
use(Web3ClientPlugin);

const posClient = new POSClient();
await posClient.init({
    network: 'testnet',
    version: 'mumbai',
    parent: {
      provider: new HDWalletProvider(privateKey, mainRPC),
      defaultConfig: {
        from : fromAddress
```

```
      }
    },
    child: {
      provider: new HDWalletProvider(privateKey, childRPC),
      defaultConfig: {
        from : fromAddress
      }
    }
});
```

After `POSClient` is initiated, we need to initiate the required token types like - `erc20` , `erc721` etc.

Let's initiate `erc20` -

ERC20

create erc20 child token

```
const erc20ChildToken = posClient.erc20(<token address>);
```

create erc20 parent token

```
const erc20ParentToken = posClient.erc20(<token address>, true);
```

Once erc20 is initaited, you can call various methods that are available, like - `getBalance` , `approve` , `deposit` , `withdraw` etc.

Let's see some of the API examples -

get balance

```
const balance = await erc20ChildToken.getBalance(<userAddress>)
console.log('balance', balance)
```

approve

```
// approve amount 10 on parent token
const approveResult = await erc20ParentToken.approve(10);

// get transaction hash
const txHash = await approveResult.getTransactionHash();

// get transaction receipt
const txReceipt = await approveResult.getReceipt();
```

As you can see, with its simple APIs maticjs makes it very easy to interact with maticjs bridge. **Let's get started with creating something awesome**

Useful links

- Examples

---

id: installation title: Installation keywords: - pos client - erc20 - withdrawExit - polygon - sdk description: Install Matic.js and Ethereum libraries.

maticjs has two parts -

1. Main library
2. Ethereum library

Main library

The Main library has the core logic and provides different APIs. The user interacts mostly with this library.

```
npm i @maticnetwork/maticjs
```

Ethereum library

Ethereum library allows us to use any favourite ether library. It is injected into maticjs using plugins.

matic.js supports two popular library -

1. Web3.js
2. Ethers

Web3.js

```
npm install @maticnetwork/maticjs-web3
```

ethers

```
npm install @maticnetwork/maticjs-ethers
```

---

id: deposit-ether title: deposit ether keywords: - 'pos client, depositEther, polygon, sdk' description: 'Deposit a required amount of ether from ethereum to polygon.'

`depositEther` method can be used to deposit required amount of **ether** from ethereum to polygon.

```
const result = await posClient.depositEther(<amount>, <userAddress>);

const txHash = await result.getTransactionHash();

const txReceipt = await result.getReceipt();
```

---

id: approve-all-for-mintable title: approveAllForMintable keywords: - 'pos client, erc115, approve, polygon, sdk' description: 'Approve ERC1155 mintable tokens.'

approveAllForMintable

`approveAllForMintable` method can be used to approve all mintable tokens on root token.

```
const erc115RootToken = posClient.erc115(<root token address>,true);

const approveResult = await erc115RootToken.approveAllForMintable();

const txHash = await approveResult.getTransactionHash();

const txReceipt = await approveResult.getReceipt();
```

---

id: approve-all title: approveAll keywords: - 'pos client, erc1155, approve, polygon, sdk' description: 'Approve ERC1155 tokens.'

approveAll

`approveAll` method can be used to approve all tokens on root token.

```
const erc1155RootToken = posClient.erc1155(<root token address>,true);

const approveResult = await erc1155RootToken.approveAll();

const txHash = await approveResult.getTransactionHash();

const txReceipt = await approveResult.getReceipt();
```

---

id: deposit-many title: depositMany keywords: - 'pos client, erc1155, deposit, polygon, sdk' description: 'Multiple deposit of ERC1155 tokens using matic.js'

`depositMany` method can be used to deposit required amounts of multiple token from ethereum to polygon chain.

```
const erc1155RootToken = posClient.erc1155(<root token address>, true);

const result = await erc1155RootToken.depositMany({
    amount: [1,2],
    tokenId: ['123','456'],
    userAddress: <from address>,
    data: '0x5465737445524331313535', // data is optional
});

const txHash = await result.getTransactionHash();

const txReceipt = await result.getReceipt();
```

Supplying data is optional.--- id: deposit title: deposit keywords: - 'pos client, erc1155, deposit, polygon, sdk' description: 'Deposit ERC1155 token using matic.js'

`deposit` method can be used to deposit required amount of a token from ethereum to polygon chain.

```
const erc1155RootToken = posClient.erc1155(<root token address>, true);

const result = await erc1155RootToken.deposit({
    amount: 1,
    tokenId: '123',
```

```
    userAddress: <from address>,
    data: '0x5465737445524331313535', // data is optional
});

const txHash = await result.getTransactionHash();

const txReceipt = await result.getReceipt();
```

Supplying data is optional.--- id: get-balance title: getBalance keywords: - 'pos client, erc1155, getBalance, polygon, sdk' description: 'Get balance of ERC1155 token using matic.js.'

`getBalance` method can be used to get the balance of user for a token. It is available on both child and parent token.

```
const erc1155Token = posClient.erc1155(<token address>);

// get balance of user
const balance = await erc1155Token.getBalance(<userAddress>, <token id>);
```

---

id: index title: POSClient keywords: - 'pos client, erc1155, contract, polygon, sdk' description: 'Interact with ERC1155 token using matic.js.'

ERC1155

`POSClient` provides `erc1155` method which helps you to interact with a erc1155 token.

The method returns instance of **ERC1155** class which contains different methods.

```
const erc721token = posClient.erc1155(<token address>, <isRoot>);
```

Passing second arguments for `isRoot` is optional.

Child token

Token on polygon can be initiated by using this syntax -

```
const childERC20Token = posClient.erc1155(<child token address>);
```

Parent token

Token on ethereum can be initiated by providing second parameter value as `true`.

```
const parentERC20Token = posClient.erc1155(<parent token address>, true);
```

---

id: is-approved-all title: isApprovedAll keywords: - 'pos client, erc1155, isApprovedAll, polygon, sdk' description: 'Check if all tokens are approved.'

`isApprovedAll` method checks if all tokens are approved for a user. It returns boolean value.

```
const erc1155Token = posClient.erc1155(<token address>, true);

const result = await erc1155Token.isApprovedAll(<user Address>);
```

---

id: is-withdraw-exited-many title: isWithdrawExitedMany keywords: - 'pos client, erc1155, isWithdrawExitedMany, polygon, sdk' description: 'Checks if withdraw has been exited for multiple tokens.'

`isWithdrawExitedMany` method check if withdraw has been exited for multiple tokens. It returns boolean value.

```
const erc1155Token = posClient.erc1155(<token address>);

const result = await erc1155Token.isWithdrawExitedMany(<exit tx hash>);
```

---

id: is-withdraw-exited title: isWithdrawExited keywords: - 'pos client, erc1155, isWithdrawExited, polygon, sdk' description: 'Checks if a withdraw has been exited.'

`isWithdrawExited` method check if a withdraw has been exited. It returns boolean value.

```
const erc1155Token = posClient.erc1155(<token address>);

const result = await erc1155Token.isWithdrawExited(<exit tx hash>);
```

---

id: transfer title: transfer keywords: - 'pos client, erc1155, transfer, polygon, sdk' description: 'Transfer tokens from one user to another user.'

`transfer` method can be used to transfer tokens from one user to another user.

```
const erc1155Token = posClient.erc1155(<token address>);

const result = await erc1155Token.transfer({
    tokenId: <tokenId>,
    amount: <amount>,
    from : <from address>,
    to : <to address>,
    data : <data to sent>, // data is optional
});

const txHash = await result.getTransactionHash();

const txReceipt = await result.getReceipt();
```

---

id: withdraw-exit-faster-many title: withdrawExitFasterMany keywords: - 'pos client, erc1155, withdrawExitFasterMany, polygon, sdk' description: 'Exit the withdraw process using txHash from withdrawStartMany.'

`withdrawExitFasterMany` method can be used to exit the withdraw process by using the txHash from `withdrawStartMany` method.

It is fast because it generates proof in backend. You need to configure setProofAPI.

Note- withdrawStart transaction must be checkpointed in order to exit the withdraw.

```
const erc1155RootToken = posClient.erc1155(<root token address>, true);

const result = await erc1155RootToken.withdrawExitFasterMany(<burn tx hash>);

const txHash = await result.getTransactionHash();

const txReceipt = await result.getReceipt();
```

---

id: withdraw-exit-faster title: withdrawExitFaster keywords: - 'pos client, erc1155, withdrawExitFaster, polygon, sdk' description: 'Exit the withdraw process using txHash from withdrawStart.'

`withdrawExitFaster` method can be used to exit the withdraw process by using the txHash from `withdrawStart` method.

It is fast because it generates proof in backend. You need to configure setProofAPI.

Note- withdrawStart transaction must be checkpointed in order to exit the withdraw.

```
const erc1155RootToken = posClient.erc1155(<root token address>, true);

const result = await erc1155RootToken.withdrawExitFaster(<burn tx hash>);

const txHash = await result.getTransactionHash();

const txReceipt = await result.getReceipt();
```

---

id: withdraw-exit-many title: withdrawExitMany keywords: - 'pos client, erc1155, withdrawExitMany, polygon, sdk' description: 'Exit the withdraw process using txHash from withdrawStart.'

`withdrawExitMany` method can be used to exit the withdraw process by using the txHash from `withdrawStartMany` method.

```
const erc1155RootToken = posClient.erc1155(<root token address>, true);

const result = await erc1155RootToken.withdrawExitMany(<burn tx hash>);

const txHash = await result.getTransactionHash();

const txReceipt = await result.getReceipt();
```

---

id: withdraw-exit title: withdrawExit keywords: - 'pos client, erc1155, withdrawExit, polygon, sdk' description: 'Exit the withdraw process using txHash from withdrawStart.'

`withdrawExit` method can be used to exit the withdraw process by using the txHash from `withdrawStart` method.

```
const erc1155RootToken = posClient.erc1155(<root token address>, true);

const result = await erc1155RootToken.withdrawExit(<burn tx hash>);

const txHash = await result.getTransactionHash();

const txReceipt = await result.getReceipt();
```

---

id: withdraw-start-many title: withdrawStartMany keywords: - 'pos client, erc1155, withdrawStartMany, polygon, sdk' description: 'Initiate the withdraw process.'

`withdrawStartMany` method can be used to initiate the withdraw process which will burn the specified amounts of multiple token respectively on polygon chain.

```
const erc1155Token = posClient.erc1155(<child token address>);

const result = await erc1155Token.withdrawStartMany([<token id1>, <token id2>],[<amount1>,<amount2>]);

const txHash = await result.getTransactionHash();

const txReceipt = await result.getReceipt();
```

---

id: withdraw-start title: withdrawStart keywords: - 'pos client, erc1155, withdrawStart, polygon, sdk' description: 'Initiate the withdraw process.'

`withdrawStart` method can be used to initiate the withdraw process which will burn the specified amount of tokenId on polygon chain.

```
const erc1155Token = posClient.erc1155(<child token address>);

const result = await erc1155Token.withdrawStart(<token id>,<amount>);

const txHash = await result.getTransactionHash();

const txReceipt = await result.getReceipt();
```

---

id: approve-max title: approveMax keywords: - pos client - erc20 - approveMax - polygon - sdk description: 'Approve max amount on the root token.'

`approveMax` method can be used to approve max amount on the root token.

```
const erc20RootToken = posClient.erc20(<root token address>, true);

const approveResult = await erc20RootToken.approveMax();

const txHash = await approveResult.getTransactionHash();

const txReceipt = await approveResult.getReceipt();
```

spenderAddress

The address on which approval is given is called `spenderAddress`. It is a third-party user or a smart contract which can transfer your token on your behalf.

By default spenderAddress value is erc20 predicate address.

You can specify spender address value manually.

```
const erc20RootToken = posClient.erc20(<root token address>,true);

// approve 100 amount
const approveResult = await erc20Token.approveMax({
    spenderAddress: <spender address value>
});

const txHash = await approveResult.getTransactionHash();

const txReceipt = await approveResult.getReceipt();
```

---

id: approve title: approve keywords: - pos client - erc20 - approve - polygon - sdk description: "Approve required amount on the root token."

`approve` method can be used to approve required amount on the root token.

approve is required in order to deposit amount on polygon chain.

```
const erc20RootToken = posClient.erc20(<root token address>,true);

// approve 100 amount
const approveResult = await erc20Token.approve(100);

const txHash = await approveResult.getTransactionHash();

const txReceipt = await approveResult.getReceipt();
```

spenderAddress

The address on which approval is given is called `spenderAddress`. It is a third-party user or a smart contract which can transfer your token on your behalf.

By default spenderAddress value is erc20 predicate address.

You can specify spender address value manually.

```
const erc20RootToken = posClient.erc20(<root token address>,true);

// approve 100 amount
const approveResult = await erc20Token.approve(100, {
    spenderAddress: <spender address value>
});

const txHash = await approveResult.getTransactionHash();

const txReceipt = await approveResult.getReceipt();

```---
id: deposit
title: deposit
keywords:
    - pos client
    - erc20
    - approveMax
    - polygon
    - sdk
description: 'Deposit required amount from root token to child token.'

`deposit` method can be used to deposit required amount from root token to child token.
```

const erc20RootToken = posClient.erc20(, true);

//deposit 100 to user address const result = await erc20Token.deposit(100, );

const txHash = await result.getTransactionHash();

const txReceipt = await result.getReceipt();

```
It might take some time to reflect the deposited amount on polygon chain. You can use [isDeposited](/docs/tools/matic-js/pos/is-deposited) method for checking status.
---
id: get-allowance
title: getAllowance
keywords:
    - pos client
    - erc20
    - getAllowance
    - polygon
    - sdk
description: "Get the approved amount for the user."
---

`getAllowance` method can be used to get the approved amount for the user.
```

const erc20Token = posClient.erc20(, true);

const balance = await erc20Token.getAllowance();

```
## spenderAddress

The address on which approval is given is called `spenderAddress`. It is a third-party user or a smart contract which can transfer your token on your behalf.

By default spenderAddress value is erc20 predicate address.
```
```

```
You can specify spender address value manually.
```

const erc20Token = posClient.erc20(, true);

const balance = await erc20Token.getAllowance(, { spenderAddress: }); ```--- id: get-balance title: getBalance keywords: - pos client - erc20 - getBalance - polygon - sdk description: "Get the balance of a user."

`getBalance` method can be used to get the balance of user. It is available on both child and parent token.

```
const erc20Token = posClient.erc20(<token address>);

// get balance of user
const balance = await erc20Token.getBalance(<userAddress>);
```

---

id: index title: POSClient keywords: - pos client - erc20 - contract - polygon - sdk description: "Provides a method to interact with an ERC20 token."

ERC20

`POSClient` provides `erc20` method which helps you to interact with an **ERC20** token.

The method returns an object which has other various methods.

```
const erc20token = posClient.erc20(<token address>,<isRoot>);
```

Passing second arguments for `isRoot` is optional.

Child token

Token on polygon can be initiated by using this syntax -

```
const childERC20Token = posClient.erc20(<child token address>);
```

Parent token

Token on ethereum can be initiated by providing the second parameter value as `true`.

```
const parentERC20Token = posClient.erc20(<parent token address>, true);
```

---

id: is-withdraw-exited title: isWithdrawExited keywords: - 'pos client, erc20, isWithdrawExited, polygon, sdk' description: 'Check whether the withdraw has been exited or not.'

`isWithdrawExited` method can be used to know whether the withdraw has been exited or not.

```
const erc20RootToken = posClient.erc20(<root token address>,true);

const isExited = await erc20Token.isWithdrawExited(<burn tx hash>);
```

---

id: transfer title: transfer keywords: - 'POS client, erc20, transfer, polygon, sdk' description: 'Transfer amount from one address to another address.'

`transfer` method can be used to transfer amount from one address to another address.

```
const erc20Token = posClient.erc20(<token address>);

const result = await erc20Token.transfer(<amount>,<to>);

const txHash = await result.getTransactionHash();

const txReceipt = await result.getReceipt();
```

---

id: withdraw-exit-faster title: withdraw exit faster keywords: - 'pos client, erc20, withdrawExitFaster, polygon, sdk' description: 'Exit the withdraw process faster using txHash from withdrawStart.'

`withdrawExitFaster` method can be used to exit the withdraw process faster by using the txHash from `withdrawStart` method.

It is generally fast because it generates proof in the backend. You need to configure setProofAPI.

Note- withdrawStart transaction must be checkpointed in order to exit the withdraw.

```
import { setProofApi } from '@maticnetwork/maticjs'

setProofApi("https://proof-generator.polygon.technology/");

const erc20RootToken = posClient.erc20(<root token address>, true);

// start withdraw process for 100 amount
const result = await erc20Token.withdrawExitFaster(<burn tx hash>);

const txHash = await result.getTransactionHash();

const txReceipt = await result.getReceipt();
```

Once the transaction is complete & checkpoint is completed, amount will be deposited to root chain.

id: withdraw-exit title: withdraw exit keywords: - 'pos client, erc20, withdrawExit, polygon, sdk' description: 'Exit the withdraw process using txHash from withdrawStart.'

`withdrawExit` method can be used to exit the withdraw process by using the txHash from `withdrawStart` method.

Note- withdrawStart transaction must be checkpointed in order to exit the withdraw.

```
const erc20RootToken = posClient.erc20(<root token address>, true);

const result = await erc20Token.withdrawExit(<burn tx hash>);

const txHash = await result.getTransactionHash();

const txReceipt = await result.getReceipt();
```

This method does multiple RPC calls to generate the proof and process exit. So it is recommended to use withdrawExitFaster method.

---

id: withdraw-start title: withdraw start keywords: - 'pos client, erc20, withdrawStart, polygon, sdk' description: 'Initiate the withdraw process.'

`withdrawStart` method can be used to initiate the withdraw process which will burn the specified amount on polygon chain.

```
const erc20Token = posClient.erc20(<child token address>);

// start withdraw process for 100 amount
const result = await erc20Token.withdrawStart(100);

const txHash = await result.getTransactionHash();

const txReceipt = await result.getReceipt();
```

The received transaction hash will be used to exit the withdraw process. So we recommend to store it.

---

id: approve-all title: approveAll keywords: - 'pos client, erc721, approveAll, polygon, sdk' description: 'Approve all tokens.'

`approveAll` method can be used to approve all tokens.

```
const erc721RootToken = posClient.erc721(<root token address>, true);

const approveResult = await erc721RootToken.approveAll();

const txHash = await approveResult.getTransactionHash();

const txReceipt = await approveResult.getReceipt();
```

---

id: approve title: approve keywords: - 'pos client, erc721, approve, polygon, sdk' description: 'Approve required amount on root token'

`approve` method can be used to approve required amount on root token.

```
const erc721RootToken = posClient.erc721(<root token address>,true);

const approveResult = await erc721RootToken.approve(<token id>);

const txHash = await approveResult.getTransactionHash();
```

```
const txReceipt = await approveResult.getReceipt();
```

---

id: deposit-many title: depositMany keywords: - 'pos client, erc721, depositMany, polygon, sdk' description: 'Deposit multiple tokens from ethereum to polygon chain.'

`depositMany` method can be used to deposit multiple token from ethereum to polygon chain.

```
const erc721RootToken = posClient.erc721(<root token address>, true);

const result = await erc721RootToken.depositMany([<token id1>,<token id2>], <user address>);

const txHash = await result.getTransactionHash();

const txReceipt = await result.getReceipt();
```

---

id: deposit title: deposit keywords: - 'pos client, erc721, deposit, polygon, sdk' description: 'Deposit a token from ethereum to polygon chain.'

`deposit` method can be used to deposit a token from ethereum to polygon chain.

```
const erc721RootToken = posClient.erc721(<root token address>, true);

const result = await erc721RootToken.deposit(<token id>, <user address>);

const txHash = await result.getTransactionHash();

const txReceipt = await result.getReceipt();
```

---

id: get-all-tokens title: getAllTokens keywords: - 'pos client, erc721, getAllTokens, polygon, sdk' description: 'Retrieve all tokens owened by specified user.'

`getAllTokens` method returns all tokens owned by specified user.

```
const erc721Token = posClient.erc721(<token address>);

const result = await erc721Token.getAllTokens(<user address>, <limit>);
```

you can also limit the tokens by specifying limit value in second parmater.

id: get-token-id-at-index-for-user title: getTokenIdAtIndexForUser keywords: - 'pos client, erc721, getTokenIdAtIndexForUser, polygon, sdk' description: 'Retrieve token id on supplied index for user.'

`getTokenIdAtIndexForUser` method returns token id on supplied index for user.

```
const erc721Token = posClient.erc721(<token address>);

const result = await erc721Token.getTokenIdAtIndexForUser(<index>,<user address>);
```

---

id: get-tokens-count title: getTokensCount keywords: - 'pos client, erc721, getTokensCount, polygon, sdk' description: 'Get tokens count for specified user.'

`getTokensCount` method returns tokens count for specified user.

```
const erc721Token = posClient.erc721(<token address>);

const result = await erc721Token.getTokensCount(<user address>);
```

---

id: index title: POSClient keywords: - 'pos client, erc721, contract, polygon, sdk' description: ' `ERC721` method which helps you to interact with a ERC721 token.'

ERC721

`POSClient` provides `erc721` method which helps you to interact with a erc721 token.

The method returns an object which has various methods.

```
const erc721token = posClient.erc721(<token address>,<isRoot>);
```

Passing second arguments for `isRoot` is optional.

Child token

Token on polygon can be initiated by using this syntax -

```
const childERC20Token = posClient.erc721(<child token address>);
```

Parent token

Token on ethereum can be initiated by providing second parameter value as `true`.

```
const parentERC20Token = posClient.erc721(<parent token address>, true);
```

---

id: is-approved-all title: isApprovedAll keywords: - 'pos client, erc721, isApprovedAll, polygon, sdk' description: 'Checks if all token are approved.'

`isApprovedAll` method checks if all token is approved. It returns boolean value.

```
const erc721Token = posClient.erc721(<token address>, true);

const result = await erc721Token.isApprovedAll(<user Address>);
```

---

id: is-approved title: isApproved keywords: - 'pos client, erc721, isApproved, polygon, sdk' description: 'Check if token is approved for a specified tokenId.'

`isApproved` method checks if token is approved for specified tokenId. It returns boolean value.

```
const erc721Token = posClient.erc721(<token address>, true);

const result = await erc721Token.isApproved(<tokenId>);
```

---

id: is-withdraw-exited-many title: isWithdrawExitedMany keywords: - 'pos client, erc721, isWithdrawExitedMany, polygon, sdk' description: 'Check if withdraw has been exited for multiple tokens.'

`isWithdrawExitedMany` method check if withdraw has been exited for multiple tokens. It returns boolean value.

```
const erc721Token = posClient.erc721(<token address>);

const result = await erc721Token.isWithdrawExitedMany(<exit tx hash>);
```

---

id: is-withdraw-exited title: isWithdrawExited keywords: - 'pos client, erc721, isWithdrawExited, polygon, sdk' description: ' Check if a withdraw has been exited.'

`isWithdrawExited` method check if a withdraw has been exited. It returns boolean value.

```
const erc721Token = posClient.erc721(<token address>);

const result = await erc721Token.isWithdrawExited(<exit tx hash>);
```

---

id: transfer title: transfer keywords: - 'pos client, erc721, transfer, polygon, sdk' description: 'Transfer tokens from one user to another user.'

`transfer` method can be used to transfer tokens from one user to another user.

```
const erc721Token = posClient.erc721(<token address>);

const result = await erc721Token.transfer(<tokenid>,<from>,<to>);

const txHash = await result.getTransactionHash();

const txReceipt = await result.getReceipt();
```

---

id: withdraw-exit-faster-many title: withdrawExitFasterMany keywords: - 'pos client, erc721, withdrawExitFasterMany, polygon, sdk' description: 'Exit the withdraw process using txHash from `withdrawStartMany`.'

`withdrawExitFasterMany` method can be used to exit the withdraw process by using the txHash from `withdrawStartMany` method.

It is fast because it generates proof in backend. You need to configure setProofAPI.

Note- withdrawStart transaction must be checkpointed in order to exit the withdraw.

```
const erc721RootToken = posClient.erc721(<root token address>, true);

const result = await erc721RootToken.withdrawExitFasterMany(<burn tx hash>);

const txHash = await result.getTransactionHash();

const txReceipt = await result.getReceipt();
```

---

id: withdraw-exit-faster title: withdrawExitFaster keywords: - 'pos client, erc721, withdrawExitFaster, polygon, sdk' description: 'Exit the withdraw process using txHash from `withdrawStart` '

`withdrawExitFaster` method can be used to exit the withdraw process by using the txHash from `withdrawStart` method.

It is fast because it generates proof in backend. You need to configure setProofAPI.

Note- withdrawStart transaction must be checkpointed in order to exit the withdraw.

```
const erc721RootToken = posClient.erc721(<root token address>, true);

const result = await erc721RootToken.withdrawExitFaster(<burn tx hash>);

const txHash = await result.getTransactionHash();

const txReceipt = await result.getReceipt();
```

---

id: withdraw-exit-many title: withdrawExitMany keywords: - 'pos client, erc721, withdrawExitMany, polygon, sdk' description: 'Exit the withdraw process using txHash from `withdrawStartMany` '

`withdrawExitMany` method can be used to exit the withdraw process by using the txHash from `withdrawStartMany` method.

```
const erc721RootToken = posClient.erc721(<root token address>, true);

const result = await erc721RootToken.withdrawExitMany(<burn tx hash>);

const txHash = await result.getTransactionHash();

const txReceipt = await result.getReceipt();
```

---

id: withdraw-exit title: withdrawExit keywords: - 'pos client, erc721, withdrawExit, polygon, sdk' description: 'Exit the withdraw process using the txHash from `withdrawStart` '

`withdrawExit` method can be used to exit the withdraw process by using the txHash from `withdrawStart` method.

```
const erc721RootToken = posClient.erc721(<root token address>, true);

const result = await erc721RootToken.withdrawExit(<burn tx hash>);

const txHash = await result.getTransactionHash();

const txReceipt = await result.getReceipt();
```

This method does multiple RPC calls to generate the proof and process exit. So it is recommended to use withdrawExitFaster method. >--- id: withdraw-start-many title: withdrawStartMany keywords: - 'pos client, erc721, withdrawStartMany, polygon, sdk' description: 'Initiate the withdraw process.'

`withdrawStartMany` method can be used to initiate the withdraw process which will burn the multiple token on polygon chain.

```
const erc721Token = posClient.erc721(<child token address>);

const result = await erc721Token.withdrawStartMany([<token id1>, <token id2>]);

const txHash = await result.getTransactionHash();

const txReceipt = await result.getReceipt();
```

---

id: withdraw-start-with-meta-data title: withdrawStartWithMetaData keywords: - 'pos client, erc721, withdrawStartWithMetaData, polygon, sdk' description: 'Initiate the withdraw process with metadata.'

`withdrawStartWithMetaData` method can be used to initiate the withdraw process which will burn the specified token on polygon chain. Under the hood it calls `withdrawWithMetadata` method on token contract.

```
const erc721Token = posClient.erc721(<token address>);

const result = await erc721Token.withdrawStartWithMetaData(<token id>);

const txHash = await result.getTransactionHash();

const txReceipt = await result.getReceipt();
```

---

id: withdraw-start title: withdrawStart keywords: - 'pos client, erc721, withdrawStart, polygon, sdk' description: 'Initiate the withdraw process.'

`withdrawStart` method can be used to initiate the withdraw process which will burn the specified token on polygon chain.

```
const erc721Token = posClient.erc721(<child token address>);

const result = await erc721Token.withdrawStart(<token id>);

const txHash = await result.getTransactionHash();

const txReceipt = await result.getReceipt();
```

---

id: index title: POSClient keywords: - 'maticjs, pos client, contract, polygon, sdk' description: 'POSClient allows you to interact with POS Bridge.'

MaticJS provides `POSClient` to interact with the **PoS Bridge**.

```
import { POSClient,use } from "@maticnetwork/maticjs"

const posClient = new POSClient();

await posClient.init({
    network: <network name>,  // 'testnet' or 'mainnet'
    version: <network version>, // 'mumbai' or 'v1'
    parent: {
      provider: <parent provider>,
      defaultConfig: {
        from: <from address>
      }
    },
    child: {
      provider: <child provider>,
      defaultConfig: {
        from: <from address>
      }
    }
});
```

Once the `POSClient` is initiated, you can interact with all available APIs.

id: is-check-pointed title: isCheckPointed keywords: - 'pos client, isCheckPointed, polygon, sdk' description: 'Check if a transaction has been checkpointed.'

`isCheckPointed` method can be used to know if a transaction has been checkpointed.

```
const isCheckPointed = await posClient.isCheckPointed(<tx hash>);
```

---

id: is-deposited title: isDeposited keywords: - 'pos client, isDeposited, polygon, sdk' description: 'Check if a deposit has been completed.'

`isDeposited` method can be used to know if a deposit has been completed.

```
const isDeposited = await posClient.isDeposited(<deposit tx hash>);
```

---

id: set-proof-api title: Configure Proof Generation API sidebar_label: Proof API keywords: - setProofApi - polygon - sdk - matic js description: Configuring the Proof Generation API, which is a publicly available proof generation API, hosted by Polygon Labs.

:::note

The API links have been changed from `https://apis.matic.network` → `https://proof-generator.polygon.technology` . Please make sure to update the links.

:::

Some of the functions in the MaticJS library are suffixed with the term `faster` . As the name suggests, they generate results more quickly compared to their non-faster counterparts. They do so by utilizing the Proof Generation API at the backend which anyone can host.

Proof Generator is a publicly available proof generation API, hosted by Polygon Labs.

The `setProofApi` method can help in setting the Proof Generation API's URL to the MaticJS instance as demonstrated in the below snippet.

```
import { setProofApi } from '@maticnetwork/maticjs'

setProofApi("https://proof-generator.polygon.technology/");
```

:::info

Utilizing a self-hosted Proof Generation API service will offer better performance compared to a publicly hosted one.

:::

Please follow the installation instructions provided in the `README.md` file in this repository to self-host the service.

Let's say you have deployed the Proof API and your base URL is `https://abc.xyz/` . In this case, you need to set the base URL in `setProofApi` as follows.

```
import { setProofApi } from '@maticnetwork/maticjs'

setProofApi("https://abc.xyz/");
```

We recommend using faster APIs because some APIs, particularly where proof is being generated, make a lot of RPC calls and it might be very slow with public RPCs.

id: ethers title: 'Ethers setup' keywords: - pos client - erc20 - withdrawExit - polygon - sdk description: 'Install and set up ethers.js'

Ether.js

The ethers.js library aims to be a complete and compact library for interacting with the Ethereum Blockchain and its ecosystem.

Setup ether.js

ether.js support is available via seperate package as a plugin for matic.js.

Installation

```
npm install @maticnetwork/maticjs-ethers
```

setup

```
import { use } from '@maticnetwork/maticjs'
import { Web3ClientPlugin } from '@maticnetwork/maticjs-ethers'

// install ethers plugin
use(Web3ClientPlugin)
```

Let's see one example of creating `POSClient` using ethers -

```
import { POSClient,use } from "@maticnetwork/maticjs"
import { Web3ClientPlugin } from '@maticnetwork/maticjs-ethers'
import { providers, Wallet } from "ethers";


// install web3 plugin
use(Web3ClientPlugin);

const parentProvider = new providers.JsonRpcProvider(rpc.parent);
const childProvider = new providers.JsonRpcProvider(rpc.child);

const posClient = new POSClient();
await posClient.init({
    network: 'testnet',
    version: 'mumbai',
    parent: {
```

```
      provider: new Wallet(privateKey, parentProvider),
      defaultConfig: {
        from : fromAddress
      }
    },
    child: {
      provider: new Wallet(privateKey, childProvider),
      defaultConfig: {
        from : fromAddress
      }
    }
});
```

Examples

The examples for different cases are available on [ethers plugin repo](#).

id: index title: 'Supported libraries ' keywords: - pos client - erc20 - withdrawExit - polygon - sdk description: 'Get started with maticjs'

Setup

In order to support multiple web3 libraries - maticjs provides plugins.

Currently two libraries are supported -

1. Web3.js
2. Ethers

---

id: web3 title: 'Web3js setup' keywords: - pos client - erc20 - withdrawExit - polygon - sdk description: 'Install and set up web3.js.'

Web3.js

web3.js is a collection of libraries that allow you to interact with a local or remote ethereum node using HTTP, IPC or WebSocket.

Setup web3.js

web3.js support is available via seperate package as a plugin for matic.js.

Installation

```
npm install @maticnetwork/maticjs-web3
```

setup

```
import { use } from '@maticnetwork/maticjs'
import { Web3ClientPlugin } from '@maticnetwork/maticjs-web3'

// install web3 plugin
use(Web3ClientPlugin)
```

Let's see an example of creating `POSClient` using web3 -

```
import { POSClient,use } from "@maticnetwork/maticjs"
import { Web3ClientPlugin } from '@maticnetwork/maticjs-web3'
import HDWalletProvider from "@truffle/hdwallet-provider"

// install web3 plugin
use(Web3ClientPlugin);

const posClient = new POSClient();
await posClient.init({
    network: 'testnet',
    version: 'mumbai',
    parent: {
      provider: new HDWalletProvider(privateKey, mainRPC),
      defaultConfig: {
        from : fromAddress
      }
    },
    child: {
      provider: new HDWalletProvider(privateKey, childRPC),
      defaultConfig: {
```

```
        from : fromAddress
      }
    }
});
```

Examples

The examples for different cases are available in [web3 plugin repo]

id: common-methods title: Miscellaneous Methods sidebar_label: Miscellaneous description: Some of the commonly used methods on the Polygon zkEVM network. keywords: - docs - maticjs - polygon - zkEvm client image: [https://wiki.polygon.technology/img/polygon-logo.png]

isDeposited

The `isDeposited` method can be used to check if a deposit has been completed.

```
const isDeposited = await zkEvmClient.isDeposited(tx_hash);
```

isDepositClaimable

The `isDepositClaimable` method checks if a deposit can be claimed on the network.

```
const isDepositClaimable = await zkEvmClient.isDepositClaimable(tx_hash);
```

isWithdrawExitable

This method checks if the withdrawal process can be exited.

```
const isWithdrawExitable = await zkEvmClient.isWithdrawExitable(tx_hash);
```

isExited

isExited method checks if a withdrawal has been exited. It returns a boolean value.

```
const isExited = await zkEvmClient.isExited(tx_hash);
```---
id: zkevm-client-erc20
title: ERC20 Methods
sidebar_label: ERC20 Methods
description: Utilize the ERC20 methods on the Polygon zkEVM network.
keywords:
  - docs
  - maticjs
  - polygon
  - zkEvm client
image: https://wiki.polygon.technology/img/polygon-logo.png

:::tip

See the [<ins>MaticJS SDK examples</ins>](https://github.com/maticnetwork/matic.js/tree/master/examples/zkEvm/erc20) on our official Github repo to interact with ERC20 tokens on the zkEVM network..

:::

Calling ERC20 Method

`ZkEvmClient` provides `erc20` method which helps you to interact with an **ERC20** token on the zkEVM network. The method returns an object which has various other methods.

```js
const erc20token = zkEvmClient.erc20(<token address> , <isRoot>);
```

Passing second argument for `isRoot` is optional.

For Child Token

Token on the zkEVM network can be initiated by using this syntax:

```
const childERC20Token = zkEvmClient.erc20(<child token address>);
```

For Root Token

Token on ethereum can be initiated by providing the **second parameter value as** `true` .

```
const rootERC20Token = zkEvmClient.erc20(<root token address>, true);
```

Check Balance

You can use the `getBalance` method to get the balance of a user account. It is available for both child and root token.

```
// get balance of user
const balance = await erc20Token.getBalance(<user Address>);
```

Approve Methods

approve

The `approve` method can approve the required amount on the root and child token. It is needed for both deposit and withdrawal on the zkEVM network. Some tokens require this method during withdrawal while others require it during deposit.

```
const erc20Token = zkEvmClient.erc20(<root token address>, true); // root token

// approve 1000 amount
const result = await erc20Token.approve(1000);

const txHash = await result.getTransactionHash();
const receipt = await result.getReceipt();
```

spenderAddress

The address on which approval is given is called the `spenderAddress` . It is a third-party user or a smart contract which can transfer your token on your behalf.

By default, `spenderAddress` value is the `PolygonZkEVMBridge` contract address. You can specify `spenderAddress` value manually.

```
// approve 1000 amount
const result = await erc20Token.approve(1000, {
    spenderAddress: <spender address value>
});
```

approveMax

The `approveMax` method can approve the maximum amount on the root and child tokens.

```
const erc20Token = zkEvmClient.erc20(<root token address>, true); // root token

const result = await erc20Token.approveMax();

const txHash = await result.getTransactionHash();
const receipt = await result.getReceipt();
```

spenderAddress

You can specify `spenderAddress` value manually.

```
// approve 100 amount
const result = await erc20Token.approveMax({
    spenderAddress: <spender address value>
});
```

isApprovalNeeded

`isApprovalNeeded` checks if approval is needed for the root or child token.

```
const erc20Token = zkEvmClient.erc20(<token address>, true); // root token

const result = await erc20Token.isApprovalNeeded();
```

getAllowance

`getAllowance` method can be used to get the approved amount for the user.

```
const erc20Token = zkEvmClient.erc20(<token address>, true); // root token
```

```
const result = await erc20Token.getAllowance(<user address>);
```

spenderAddress

You can specify spender address value manually.

```
const result = await erc20Token.getAllowance(<user Address>, {
    spenderAddress: <spender address value>
});
```

Transfer Method

The `transfer` method can be used to transfer amount from one address to another.

```
const erc20Token = zkEvmClient.erc20(<token address>);

const result = await erc20Token.transfer(<amount>, <to>);

const txHash = await result.getTransactionHash();
const receipt = await result.getReceipt();
```

Deposit Methods

deposit

`deposit` method can be used to deposit the required amount from root chain to the child chain. We recommend users to store the transaction hash in order to be able to call `depositClaim` using that `txHash`.

```
const erc20Token = zkEvmClient.erc20(<root token address>, true); // root token

//deposit 100 to user address
const result = await erc20Token.deposit(100, <user address>);

const txHash = await result.getTransactionHash();
const receipt = await result.getReceipt();
```

depositEther

`depositEther` method can be used to deposit required amount of **ether** from Ethereum to zkEVM.

```
// ether address = 0x0000000000000000000000000000000000000000
const etherToken = zkEvmClient.erc20(<ether address>, true);

const result = await etherToken.deposit(<amount>, <user Address>);

const txHash = await result.getTransactionHash();
const receipt = await result.getReceipt();
```

depositWithPermit

`depositWithPermit` method can be used to deposit required amount of tokens from Ethereum to zkEVM along with the permit, so that user doesn't have to do multiple transactions for `approve` and `deposit`.

```
const erc20Token = zkEvmClient.erc20(<root token address>, true); // root token

const result = await erc20Token.depositWithPermit(<amount>, <user address>);

const txHash = await result.getTransactionHash();
const receipt = await result.getReceipt();
```

depositClaim

`depositClaim` method is used for child tokens to claim their ERC20 token deposits.

```
const erc20Token = zkEvmClient.erc20(<child token address>); // child token

const result = await erc20Token.depositClaim(<transaction hash);

const txHash = await result.getTransactionHash();
const receipt = await result.getReceipt();
```

Token Withdrawal Methods

withdraw

`withdraw` method can be used to initiate the withdrawal process which transfers tokens from zkEVM network to Ethereum.

```
const erc20Token = zkEvmClient.erc20(<child token address>); // child token

const result = await erc20Token.withdraw(<amount>, <user address);

const txHash = await result.getTransactionHash();
const receipt = await result.getReceipt();
```

The received transaction hash will be used to exit the withdraw process. So we recommend to store it.

withdrawExit

`withdrawExit` method can be used to exit the withdrawal process by using the transaction hash from `withdraw` method. Note that the validity proof of `withdraw` transaction must be submitted in order to exit the withdrawal process.

```
const erc20Token = zkEvmClient.erc20(<root token address>, true); // root token

const result = await erc20Token.withdrawExit(<transaction hash>);

const txHash = await result.getTransactionHash();
const receipt = await result.getReceipt();
```

---

id: initialize-zkevm title: Initialize ZkEvmClient sidebar_label: Initialize description: ZkEvmClient allows you to interact with the Polygon zkEVM network. keywords: - docs - maticjs - polygon - zkEvm client image: https://wiki.polygon.technology/img/polygon-logo.png

MaticJS library also provides **ZkEvmClient** to interact with the Polygon zkEVM network.

```
import { ZkEvmClient, use } from "@maticnetwork/maticjs"

const zkEvmClient = new ZkEvmClient();

await zkEvmClient.init({
  network: <network name>,  // 'testnet'
  version: <network version>, // 'blueberry'
  parent: {
    provider: <parent provider>,
    defaultConfig: {
      from: <from address>
    }
  },
  child: {
    provider: <child provider>,
    defaultConfig: {
      from: <from address>
    }
  }
});
```

Once the `ZkEvmClient` is initialized, you can interact with all available APIs from MaticJS SDK.

id: api3 title: API3 sidebar_label: API3 description: API3 is a first-party oracle that delivers more security, efficiency, and data-source transparency. keywords: - wiki - polygon - api3 - oracle - off-chain data image: https://wiki.polygon.technology/img/polygon-wiki.png

import useBaseUrl from '@docusaurus/useBaseUrl';

:::caution Content disclaimer

Please view the third-party content disclaimer here.

:::

Overview

API3 is a collaborative project to **deliver traditional API services to smart contract platforms** in a decentralized and trust-minimized way. It is governed by a decentralized autonomous organization (DAO), namely the API3 DAO.

First-party oracles

An Airnode is a **first-party oracle** that pushes off-chain API data to your on-chain contract. Airnode lets API providers easily run their own oracle nodes. That way, they can provide data to any on-chain dApp that's interested in their services, all without an intermediary.

An on-chain smart contract makes a request in the **RRP protocol contract (AirnodeRrpV0.sol)** that adds the request to the event logs. The Airnode then accesses the event logs, fetches the API data and performs a callback to the requester with the requested data.

> *<img src={useBaseUrl("img/api3/airnode1.png")} width="700"/>*

Requesting off-chain data by calling an Airnode

Requesting off-chain data essentially involves triggering an Airnode and getting its response through your smart contract. The smart contract in this case would be the requester contract which will make a request to the desired off-chain Airnode and then capture its response.

The requester calling an Airnode primarily focuses on two tasks:

- **Make the request**
- **Accept and decode the response**

> *<img src={useBaseUrl("img/api3/airnode2.png")} width="600"/>*

Here is an example of a basic requester contract to request data from an Airnode:

```
pragma solidity 0.8.9;

import "@api3/airnode-protocol/contracts/rrp/requesters/RrpRequesterV0.sol";

// A Requester that will return the requested data by calling the specified airnode.
// Make sure you specify the right _rrpAddress for your chain.

contract Requester is RrpRequesterV0 {
    mapping(bytes32 => bool) public incomingFulfillments;
    mapping(bytes32 => int256) public fulfilledData;

    constructor(address _rrpAddress) RrpRequesterV0(_rrpAddress) {}

    /**
     * The main makeRequest function that will trigger the Airnode request
     * airnode: Airnode address
     * endpointId: The endpoint ID for the specific endpoint
     * sponsor: The requester contract itself (in this case)
     * sponsorWallet: The wallet that will make the actual request (needs to be funded)
     * parameters: encoded API parameters
     */
    function makeRequest(
        address airnode,
        bytes32 endpointId,
        address sponsor,
        address sponsorWallet,
        bytes calldata parameters

    ) external {
        bytes32 requestId = airnodeRrp.makeFullRequest(
            airnode,
            endpointId,
            sponsor,
            sponsorWallet,
            address(this),
            this.fulfill.selector,
            parameters
        );
        incomingFulfillments[requestId] = true;
    }

    // The callback function with the requested data
    function fulfill(bytes32 requestId, bytes calldata data)
        external
        onlyAirnodeRrp
    {
        require(incomingFulfillments[requestId], "No such request made");
        delete incomingFulfillments[requestId];
        int256 decodedData = abi.decode(data, (int256));
        fulfilledData[requestId] = decodedData;
    }
}
```

The `_rrpAddress` is the main `airnodeRrpAddress`. The RRP Contracts have already been deployed on-chain. You can check the address for Polygon here. You can also try **deploying it on Remix**

Contract	Addresses

AirnodeRrpV0	`0xa0AD79D995DdeeB18a14eAef56A549A04e3Aa1Bd`

Request parameters

The `makeRequest()` function expects the following parameters to make a valid request.

- **`airnode`** : Specifies the Airnode Address.
- **`endpointId`** : Specifies which endpoint to be used.
- **`sponsor`** and **`sponsorWallet`** : Specifies which wallet will be used to fulfill the request.
- **`parameters`** : Specifies the API and Reserved Parameters (see Airnode ABI specifications for how these are encoded). Parameters can be encoded off-chain using `@airnode-abi` library.

Response parameters

The callback to the **Requester** contains two parameters:

- `requestId` : First acquired when making the request and passed here as a reference to identify the request for which the response is intended.
- `data` : In case of a successful response, this is the requested data which has been encoded and contains a timestamp in addition to other response data. Decode it using the `decode()` function from the `abi` object.

:::note Sponsors should not fund a `sponsorWallet` with more then they can trust the Airnode with, as the Airnode controls the private key to the `sponsorWallet` . The deployer of such Airnode undertakes no custody obligations, and the risk of loss or misuse of any excess funds sent to the `sponsorWallet` remains with the sponsor :::

Try deploying it on Remix!

Using dAPIs - API3 Datafeeds

dAPIs are continuously updated streams of off-chain data, such as the latest cryptocurrency, stock and commodity prices. They can power various decentralized applications such as DeFi lending, synthetic assets, stablecoins, derivatives, NFTs and more.

The data feeds are continuously updated by first-party oracles using signed data. dApp owners can read the on-chain value of any dAPI in realtime.

Due to being composed of first-party data feeds, dAPIs offer security, transparency, cost-efficiency and scalability in a turn-key package.

The API3 Market enables users to connect to a dAPI and access the associated data feed services.

> *<img src={useBaseUrl("img/api3/SS4.png")} width="700"/>*

To know more about how dAPIs work, click here

Subscribing to self-funded dAPIs

With self-funded dAPIs, you can fund the dAPI with your own funds. The amount of gas you supply will determine how long your dAPI will be available for use. If you run out of gas, you can fund the dAPI again to keep it available for use.

Exploring and selecting your dAPI

The API3 Market provides a list of all the dAPIs available across multiple chains including testnets. You can filter the list by chains and data providers. You can also search for a specific dAPI by name. Once selected you will land on the details page where you can find more information about the dAPI.

Funding a sponsor wallet

Once you have selected your dAPI, you can activate it by using the API3 Market to send Matic to the `sponsorWallet` . Make sure your:

- Wallet is connected to the Market and is the same network as the dAPI you are funding.
- Balance of the wallet should be greater than the amount you are sending to the `sponsorWallet` .

> *<img src={useBaseUrl("img/api3/market1.png")} width="500"/>*

To fund the dAPI, you need to click on the **Fund Gas** button. Depending upon if a proxy contract is already deployed, you will see a different UI.

> *<img src={useBaseUrl("img/api3/market2.png")} width="700"/>*

Use the gas estimator to select how much gas is needed by the dAPI. Click on **Send Matic** to send the entered amount to the sponsor wallet of the respective dAPI.

> *<img src={useBaseUrl("img/api3/market3.png")} width="500"/>*

Once the transaction is broadcasted & confirmed on the blockchain a transaction confirmation screen will appear.

> *<img src={useBaseUrl("img/api3/market4.png")} width="500"/>*

Deploying a proxy contract to access the dAPI

Smart contracts can interact and read values from contracts that are already deployed on the blockchain. By deploying a proxy contract via the API3 Market, a dApp can interact and read values from a dAPI like ETH/USD.

:::info Note If a proxy is already deployed for a self-funded dAPI, the dApp can read the dAPI without having to deploy a proxy contract. They do this by using the address of the already deployed proxy contract which will be visible on the API3 Market. :::

If you are deploying a proxy contract during the funding process, clicking on the **Get proxy** button will initiate a transaction to your MetaMask that will deploy a proxy contract.

> *<img src={useBaseUrl("img/api3/market5.png")} width="400"/>*

Once the transaction is broadcasted and confirmed on the blockchain, the proxy contract address will be shown on the UI.

> *<img src={useBaseUrl("img/api3/market6.png")} width="500"/>*

Reading from a self-funded dAPI

Here's an example of a basic contract that reads from a self-funded dAPI.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.17;

import "@openzeppelin/contracts/access/Ownable.sol";
import "@api3/contracts/v0.8/interfaces/IProxy.sol";

contract DataFeedReaderExample is Ownable {
    // This contract reads from a single proxy. Your contract can read from multiple proxies.
    address public proxy;

    // Updating the proxy address is a security-critical action. In this example, only
    // the owner is allowed to do so.
    function setProxy(address _proxy) public onlyOwner {
        proxy = _proxy;
    }

    function readDataFeed()
        external
        view
        returns (int224 value, uint256 timestamp)
    {
        (value, timestamp) = IProxy(proxy).read();
        // If you have any assumptions about `value` and `timestamp`, make sure
        // to validate them right after reading from the proxy.
    }
}
```

- `setProxy()` is used to set the address of the dAPI Proxy Contract.
- `readDataFeed()` is a view function that returns the latest price of the set dAPI.

You can read more about dAPIs here.

Try deploying it on Remix!

Using API3 QRNG

API3 QRNG is a public utility we provide with the courtesy of Australian National University (ANU). It is powered by an Airnode hosted by ANU Quantum Random Numbers, meaning that it is a first-party service. It is served as a public good and is free of charge (apart from the gas costs), and it **provides 'true' quantum randomness** via an easy-to-use solution when requiring RNG on-chain.

To request randomness on-chain, the requester submits a request for a random number to `AirnodeRrpV0`. The ANU Airnode gathers the request from the `AirnodeRrpV0` protocol contract, retrieves the random number off-chain, and sends it back to `AirnodeRrpV0`. Once received, it performs a callback to the requester with the random number.

QRNG Getting Started

Here is an example of a basic `QrngRequester` that requests a random number:

```
//SPDX-License-Identifier: MIT
pragma solidity 0.8.9;
import "@api3/airnode-protocol/contracts/rrp/requesters/RrpRequesterV0.sol";

contract RemixQrngExample is RrpRequesterV0 {
    event RequestedUint256(bytes32 indexed requestId);
    event ReceivedUint256(bytes32 indexed requestId, uint256 response);

    address public airnode;
    bytes32 public endpointIdUint256;
    address public sponsorWallet;
    mapping(bytes32 => bool) public waitingFulfillment;

    // These are for Remix demonstration purposes, their use is not practical.
    struct LatestRequest {
      bytes32 requestId;
      uint256 randomNumber;
    }
    LatestRequest public latestRequest;

    constructor(address _airnodeRrp) RrpRequesterV0(_airnodeRrp) {}
```

```
    // Normally, this function should be protected, as in:
    // require(msg.sender == owner, "Sender not owner");
    function setRequestParameters(
        address _airnode,
        bytes32 _endpointIdUint256,
        address _sponsorWallet
    ) external {
        airnode = _airnode;
        endpointIdUint256 = _endpointIdUint256;
        sponsorWallet = _sponsorWallet;
    }

    function makeRequestUint256() external {
        bytes32 requestId = airnodeRrp.makeFullRequest(
            airnode,
            endpointIdUint256,
            address(this),
            sponsorWallet,
            address(this),
            this.fulfillUint256.selector,
            ""
        );
        waitingFulfillment[requestId] = true;
        latestRequest.requestId = requestId;
        latestRequest.randomNumber = 0;
        emit RequestedUint256(requestId);
    }

    function fulfillUint256(bytes32 requestId, bytes calldata data)
        external
        onlyAirnodeRrp
    {
        require(
            waitingFulfillment[requestId],
            "Request ID not known"
        );
        waitingFulfillment[requestId] = false;
        uint256 qrngUint256 = abi.decode(data, (uint256));
        // Do what you want with `qrngUint256` here...
        latestRequest.randomNumber = qrngUint256;
        emit ReceivedUint256(requestId, qrngUint256);
    }
}
```

- The `setRequestParameters()` takes in `airnode` (The ANU/Quintessence/Nodary Airnode address) , `endpointIdUint256` , `sponsorWallet` and sets these parameters. You can get Airnode address and the endpoint ID here.
- The `makeRequestUint256()` function calls the `airnodeRrp.makeFullRequest()` function of the `AirnodeRrpV0.sol` protocol contract which adds the request to its storage and returns a `requestId` .
- The targeted off-chain ANU Airnode gathers the request and performs a callback to the requester with the random number.

You can read more about API3 QRNG here.

Try deploying it on Remix!

ChainAPI

ChainAPI is the Web3 data integration platform that provides businesses with all the tools they need to connect their data to blockchain-based applications. API providers can follow a simple GUI-based integration and deployment flow and start running their first-party Airnode without writing a single line of code.

If you have a REST API, you can easily deploy your own Airnode using ChainAPI to Polygon.

Check out the API3 Docs here.

Additional Resources

Here are some additional developer resources

- API3 Docs
- dAPI Docs
- QRNG Docs
- Github
- Medium--- id: bandchain title: BandChain sidebar_label: BandChain description: BandChain is a High-performance Blockchain Built for Data Oracle to query data from traditional web APIs keywords:
    - wiki
    - polygon
    - oracles
    - bandchain

- web apis
- band protocol image: https://wiki.polygon.technology/img/polygon-logo.png

---

import useBaseUrl from '@docusaurus/useBaseUrl';

:::caution Content disclaimer

Please view the third-party content disclaimer here.

:::

Band Protocol allows you to query data from traditional web APIs and use it in the blockchain. Developers can make queries through **BandChain, a cosmos-based blockchain** for facilitating oracle requests and payment, and then use the data on the dApp through inter-chain communication. Integrating oracle data can be done in 3 simple steps:

1. **Choosing the oracle scripts**

   Oracle script is a hash that uniquely identifies the type of data to be requested from band-chain. These scripts can be found **here**. These scripts are used as one of the parameters while making the oracle request.

2. **Requesting Data from BandChain**

   This can be done in two ways:

   - **Using the BandChain explorer**

     You can click on the oracle script of your choice, and then from the **Execute** tab you can pass in the parameters and get the response from BandChain. The response will contain the result and also an EVM proof. This proof has to be copied and will be used in the final step. The BandChain docs for querying oracle using explorer is available **here**.

     <img src={useBaseUrl("img/bandchain/executeoracle.png")} />

     Given above is an example of making an oracle request to get the random number values. The value 100 is passed to the `max_range` parameter of the oracle request. We get a hash in response. Clicking on this hash will show us the complete details of the response.

   - **Using the BandChain-Devnet JS Library**

     You can query BandChain directly using the BandChain-Devnet library. When queried, it gives an **EVM proof** in the response. This proof can be used for the final step of BandChain integration. The BandChain docs for querying oracle using BandChain-Devnet JS Library is available **here**. The request payload for the random number oracle will look like this. Make sure the request body is passed in application/json format.

3. **Using the data in Smart Contracts**

The final step is to deploy a validation contract and store the responses from the oracle request into the validation contracts state variables. Once these state variables are set, they can be accessed as and when required by the dApp. Also these state variables can be updated with new values by querying the oracle scripts again from the dApp. Given below is a validation contract that stores the random number value using the random number oracle script.

```
pragma solidity 0.5.14;
pragma experimental ABIEncoderV2;

import "BandChainLib.sol";
import "IBridge.sol";

contract SimplePriceDatabase {
  using BandChainLib for bytes;

  bytes32 public codeHash;
  bytes public params;
  IBridge public bridge;

  uint256 public latestPrice;
  uint256 public lastUpdate;

  constructor(
    bytes32 _codeHash ,
    bytes memory _params,
    IBridge _bridge
  ) public {
    codeHash = _codeHash;
    params = _params;
    bridge = _bridge;
  }

  function update(bytes memory _reportPrice) public {
    IBridge.VerifyOracleDataResult memory result = bridge.relayAndVerify(_reportPrice);
    uint64[] memory decodedInfo = result.data.toUint64List();

    require(result.codeHash == codeHash, "INVALID_CODEHASH");
    require(keccak256(result.params) == keccak256(params), "INVALID_PARAMS");
    require(uint256(decodedInfo[1]) > lastUpdate, "TIMESTAMP_MUST_BE_OLDER_THAN_THE_LAST_UPDATE");
```

```
    latestPrice = uint256(decodedInfo[0]);
    lastUpdate = uint256(decodedInfo[1]);
  }
}
```

When deploying, 3 parameters have to be passed. The **first parameter** is the `codeHash` which is the oracle script hash. The **second parameter** is the oracle script request parameters object. This has to be passed in bytes format. BandChain provides a REST API for converting the parameter JSON object to bytes format. The API details can be found **here**. A 0x has to be appended to the response received from this API. The **third parameter** is the contract address of the BandChain contract that is already deployed on the Polygon network. Band Protocol supports Polygon TestnetV3: 0x3ba819b03fb8d34995f68304946eefa6dcff7cbf.

Another thing to note is that the validation contract should import the helper library and interface which is called `BandChainLib.sol` and `IBridge.sol` respectively. They can be found in the following links: **Bandchain** Library and **IBridge** interface.

Once the validation contract is deployed, the state variables can be accessed by querying from a dApp. Similarly multiple validation contracts can be created for different in-built oracle scripts. The IBridge interface has a method called `relayAndVerify()` that verifies the values being updated each time in the validation contract. The `update()` method in the validation contract has the logic to update the state variables. The EVM proof obtained from querying the oracle script has to be passed to the `update()` method. Each time a value is updated, the BandChain contract deployed on Polygon verifies the data before storing it in the contract state variable.

The BandChain provides a decentralized network of oracles that can be used by dApps to boost their smart contract logic. The BandChain docs on deploying the contract, storing the values, and updating them can be found here.--- id: bandstandarddataset title: Band Standard Dataset sidebar_label: Standard Dataset description: Band Stardard Dataset offers real-time price information for over 196+ symbols spanning across crypto assets, foreign exchange and commodities keywords: - wiki - polygon - oracles - bandchain - web apis - standard dataset - band protocol image: https://wiki.polygon.technology/img/polygon-logo.png

import useBaseUrl from '@docusaurus/useBaseUrl';

:::caution Content disclaimer

Please view the third-party content disclaimer here.

:::

Developers building on Polygon can now leverage Band Protocol's decentralized oracle infrastructure. With Band Protocol's oracle, they now have access to various cryptocurrency price data to integrate into their applications.

Supported Tokens

Currently, the list of supported symbols can be found at data.bandprotocol.com. Going forward, this list will continue to expand based on developer needs and community feedback.

Price Pairs

The following methods can work with any combination of base/quote token pair, as long as the base and quote symbols are supported by the dataset.

Querying Prices

Currently, there are two methods for developers to query prices from Band Protocol's oracle: through Band's `StdReference` smart contract on Polygon and through their `bandchain.js` JavaScript helper library.

Solidity Smart Contract

To query prices from Band Protocol's oracle, a smart contract should reference Band's `StdReference` contract, specifically the `getReferenceData` and `getReferenceDatabulk` methods.

`getReferenceData` takes two strings as the inputs, the `base` and `quote` symbol, respectively. It then queries the `StdReference` contract for the latest rates for those two tokens, and returns a `ReferenceData` struct, shown below.

```
struct ReferenceData {
    uint256 rate; // base/quote exchange rate, multiplied by 1e18.
    uint256 lastUpdatedBase; // UNIX epoch of the last time when base price gets updated.
    uint256 lastUpdatedQuote; // UNIX epoch of the last time when quote price gets updated.
}
```

`getReferenceDataBulk` instead takes two lists, one of the `base` tokens, and one of the `quotes`. It then proceeds to similarly query the price for each base/quote pair at each index, and returns an array of `ReferenceData` structs.

For example, if we call `getReferenceDataBulk` with `['BTC','BTC','ETH']` and `['USD','ETH','BNB']`, the returned `ReferenceData` array will contain information regarding the pairs:

- `BTC/USD`
- `BTC/ETH`
- `ETH/BNB`

Contract Addresses

Blockchain	Contract Address
Polygon (Test)	`0x56e2898e0ceff0d1222827759b56b28ad812f92f`

BandChain.JS

Band's node helper library `bandchain.js` also supports a similar `getReferenceData` function. This function takes one argument, a list of token pairs to query the result. It then returns a list of corresponding rate values.

Example Usage

The code below shows an example usage of the function:

```
const { Client } = require('@bandprotocol/bandchain.js');

// BandChain's REST Endpoint
const endpoint = 'https://rpc.bandchain.org';
const client = new Client(endpoint);

// This example demonstrates how to query price data from
// Band's standard dataset
async function exampleGetReferenceData() {
  const rate = await client.getReferenceData(['BTC/ETH','BAND/EUR']);
  return rate;
}

(async () => {
  console.log(await exampleGetReferenceData());
})();
```

The corresponding result will then be similar to:

```
$ node index.js
[
    {
        pair: 'BTC/ETH',
        rate: 30.998744363906173,
        updatedAt: { base: 1615866954, quote: 1615866954 },
        requestID: { base: 2206590, quote: 2206590 }
    },
    {
        pair: 'BAND/EUR',
        rate: 10.566138918332376,
        updatedAt: { base: 1615866845, quote: 1615866911 },
        requestID: { base: 2206539, quote: 2206572 }
    }
]
```

For each pair, the following information will be returned:

- `pair` : The base/quote symbol pair string
- `rate` : The resulting rate of the given pair
- `updated` : The timestamp at which the base and quote symbols was last updated on BandChain. For `USD` , this will be the current timestamp.
- `rawRate` : This object consists of two parts.
  - `value` is the `BigInt` value of the actual rate, multiplied by `10^decimals`
  - `decimals` is then the exponent by which `rate` was multiplied by to get `rawRate`

Example Usage

This contract demonstrates an example of using Band's `StdReference` contract and the `getReferenceData` function.--- id: chainlink title: Chainlink sidebar_label: Chainlink description: Chainlink is a decentralized blockchain oracle network built on Ethereum. keywords: - wiki - polygon - chainlink - oracle - decentralized - data image: https://wiki.polygon.technology/img/polygon-logo.png

import useBaseUrl from '@docusaurus/useBaseUrl';

:::caution Content disclaimer

Please view the third-party content disclaimer here.

:::

Chainlink enables your contracts to **access any external data source**, through a decentralized oracle network. Whether your contract requires sports results, the latest weather, or any other publicly available data, Chainlink provides the tools required for your contract to consume it.

Decentralized Data

One of Chainlink's most powerful features is already decentralized, aggregated, and ready to be digested on-chain data on most of the popular cryptocurrencies. These are known as **Chainlink Data Feeds**.

Here is a working example of a contract that pulls the latest price of MATIC in USD on the Mumbai Testnet.

All you need to do is swap out the address with any address of a data feed that you wish, and you can start digesting price information.

```
pragma solidity ^0.6.7;

import "@chainlink/contracts/src/v0.6/interfaces/AggregatorV3Interface.sol";

contract PriceConsumerV3 {
    AggregatorV3Interface internal priceFeed;

    /**
     * Network: Mumbai Testnet
     * Aggregator: MATIC/USD
     * Address: 0xd0D5e3DB44DE05E9F294BB0a3bEEaF030DE24Ada
     */
    constructor() public {
        priceFeed = AggregatorV3Interface(0xd0D5e3DB44DE05E9F294BB0a3bEEaF030DE24Ada);
    }

    /**
     * Returns the latest price
     */
    function getLatestPrice() public view returns (int) {
        (
            uint80 roundID,
            int price,
            uint startedAt,
            uint timeStamp,
            uint80 answeredInRound
        ) = priceFeed.latestRoundData();
        return price;
    }
}
```

Request and Receive Cycle

Chainlink's Request and Receive cycle enables your smart contracts to make a request to any external API and consume the response. To implement it, your contract needs to define two functions:

1. One to **request the data**, and
2. Another to **receive the response**.

To request data, your contract builds a `request` object which it provides to an oracle. Once the oracle has reached out to the API and parsed the response, it will attempt to send the data back to your contract using the callback function defined in your smart contract.

Uses

1. **Chainlink Data Feeds**

   These are decentralized data reference points already aggregated on-chain, and the quickest, easiest, and cheapest way to get data from the real world. Currently supports some of the most popular cryptocurrency and fiat pairs.

   For working with Data Feeds, use the **Polygon Data Feeds** from the Chainlink documenation.

2. **Chainlink Verifiable Randomness Function**

   Get provably random numbers, where the random number is cryptographically guaranteed to be random.

   For working with Chainlink VRF, use the **Polygon VRF** addresses from the Chainlink documentation.

3. **Chainlink API Calls**

   How to configure your smart contract to work with traditional APIs, and customize to get any data, send any requests over the internet, and more.

Code Example

To interact with external APIs, your smart contract should inherit from `ChainlinkClient.sol`, which is a contract designed to make processing requests easy. It exposes a struct called `Chainlink.Request`, which your contract should use to build the API request.

The request should define the oracle address, job id, fee, adapter parameters, and the callback function signature. In this example, the request is built in the `requestEthereumPrice` function.

`fulfill` is defined as the callback function.

```
pragma solidity ^0.6.0;

import "@chainlink/contracts/src/v0.6/ChainlinkClient.sol";

contract APIConsumer is ChainlinkClient {

    uint256 public price;

    address private oracle;
    bytes32 private jobId;
    uint256 private fee;

    /**
     * Network: Polygon Mumbai Testnet
     * Oracle: 0x58bbdbfb6fca3129b91f0dbe372098123b38b5e9
     * Job ID: da20aae0e4c843f6949e5cb3f7cfe8c4
     * LINK address: 0x326C977E6efc84E512bB9C30f76E30c160eD06FB
     * Fee: 0.01 LINK
     */
    constructor() public {
        setChainlinkToken(0x326C977E6efc84E512bB9C30f76E30c160eD06FB);
        oracle = 0x58bbdbfb6fca3129b91f0dbe372098123b38b5e9;
        jobId = "da20aae0e4c843f6949e5cb3f7cfe8c4";
        fee = 10 ** 16; // 0.01 LINK
    }

    /**
     * Create a Chainlink request to retrieve API response, find the target price
     * data, then multiply by 100 (to remove decimal places from price).
     */
    function requestBTCCNYPrice() public returns (bytes32 requestId)
    {
        Chainlink.Request memory request = buildChainlinkRequest(jobId, address(this), this.fulfill.selector);

        // Set the URL to perform the GET request on
        // NOTE: If this oracle gets more than 5 requests from this job at a time, it will not return.
        request.add("get", "https://www.alphavantage.co/query?
function=CURRENCY_EXCHANGE_RATE&from_currency=BTC&to_currency=CNY&apikey=demo");

       // Set the path to find the desired data in the API response, where the response format is:
       // {
       //     "Realtime Currency Exchange Rate": {
       //       "1. From_Currency Code": "BTC",
       //       "2. From_Currency Name": "Bitcoin",
       //       "3. To_Currency Code": "CNY",
       //       "4. To_Currency Name": "Chinese Yuan",
       //       "5. Exchange Rate": "207838.88814500",
       //       "6. Last Refreshed": "2021-01-26 11:11:07",
       //       "7. Time Zone": "UTC",
       //      "8. Bid Price": "207838.82343000",
       //       "9. Ask Price": "207838.88814500"
       //     }
       //     }
        string[] memory path = new string[](2);
        path[0] = "Realtime Currency Exchange Rate";
        path[1] = "5. Exchange Rate";
        request.addStringArray("path", path);

        // Multiply the result by 10000000000 to remove decimals
        request.addInt("times", 10000000000);

        // Sends the request
        return sendChainlinkRequestTo(oracle, request, fee);
    }

    /**
     * Receive the response in the form of uint256
     */
    function fulfill(bytes32 _requestId, uint256 _price) public recordChainlinkFulfillment(_requestId)
    {
        price = _price;
    }
}
```

Mainnet Polygon LINK token

To get mainnet Polygon LINK token from the Ethereum Mainnet, you must follow a 2-step process.

1. Bridge your LINK using the PoS bridge.
2. Swap the LINK for the ERC677 version via the Pegswap, deployed by the Chainlink.

The Polygon bridge brings over an ERC20 version of LINK, and LINK is an ERC677, so we just have to update it with this swap.

Addresses

There are currently only a few operational Chainlink oracles on the Polygon Mumbai Testnet. You can always run one yourself too, and list it on the Chainlink Marketplace.

- Oracle: `0xb33D8A4e62236eA91F3a8fD7ab15A95B9B7eEc7D`
- LINK: `0x326C977E6efc84E512bB9C30f76E30c160eD06FB`

To obtain LINK on Mumbai Testnet, head to the Polygon faucet here.

Supported APIs

Chainlink's Request and Receive cycle is flexible enough to call any public API, so long as the request parameters are correct and the response format is known. For example, if the response object from a URL we want to fetch from is formatted like this: `{"USD":243.33}` , the path is simple: `"USD"` .

If an API responds with a complex JSON object, the **path** parameter would need to specify where to retrieve the desired data, using a dot delimited string for nested objects. For example, consider the following response:

```
{
   "Prices":{
        "USD":243.33
    }
}
```

This would require the following path: `"Prices.USD"` . If there are spaces in the strings, or the strings are quite long, we can use the syntax shown in the example above, where we pass them all as a string array.

```
string[] memory path = new string[](2);
path[0] = "Prices";
path[1] = "USD";
request.addStringArray("path", path);
```

What Are Job IDs For?

You may have noticed that our example uses a `jobId` parameter when building the request. Jobs are comprised of a sequence of instructions that an oracle is configured to run. In the code example above, the contract makes a request to the oracle with the job ID: `da20aae0e4c843f6949e5cb3f7cfe8c4` . This particular job is configured to do the following:

- Make a GET request
- Parse the JSON response
- Multiply the value by *x*
- Convert the value to `uint`
- Submit to the chain

This is why our contract adds in the URL, the path of where to find the desired data in the JSON response, and the times amount to the request; using the `request.add` statements. These instructions are facilitated by what's known as Adapters, in the oracle.

Every request to an oracle must include a specific job ID.

Here is the list of jobs that the Polygon oracle is configured to run.

Name	Return Type	ID	Adapters

HTTP GET	`uint256`	`da20aae0e4c843f6949e5cb3f7cfe8c4`	`httpget`
			`jsonparse`
			`multiply`
			`ethuint256`
			`ethtx`

HTTP GET	`int256`	`e0c76e45462f4e429ba32c114bfbf5ac`	`httpget`
			`jsonparse`
			`multiply`
			`ethint256`
			`ethtx`

HTTP GET	`bool`	`999539ec63414233bdc989d8a8ff10aa`	`httpget`
			`jsonparse`
			`ethbool`
			`ethtx`

HTTP GET	`bytes32`	`a82495a8fd5b4cb492b17dc0cc31a4fe`	`httpget` `jsonparse`

			`ethbytes32` `ethtx`

HTTP GET	`string`	`7d80a6386ef543a3abb52817f6707e3b`	`httpget`
			`jsonparse`
			`ethstring`
			`ethtx`

HTTP POST	`bytes32`	`a82495a8fd5b4cb492b17dc0cc31a4fe`	`httppost`
			`jsonparse`
			`ethbytes32`
			`ethtx`

The complete Chainlink API reference can be found here.

id: diadataoracles title: DIA Data description: Build your next blockchain app on Polygon. keywords: - docs - matic - dia - oracle image: https://wiki.polygon.technology/img/polygon-logo.png

import useBaseUrl from '@docusaurus/useBaseUrl';

:::caution Content disclaimer

Please view the third-party content disclaimer here.

:::

DApp developers who want to leverage DIA oracles can access the published data on Polygon. DIA offers cryptocurrency and data about traditional financial assets. For more details click here--- id: getting-started title: Getting Started sidebar_label: Getting Started description: Available solutions for fetching off-chain data into Polygon dApps keywords: - wiki - polygon - data oracles - chainlink - bandchain - api3 image: https://wiki.polygon.technology/img/polygon-logo.png

There is often a need to access information from the outside world that is relevant to the contractual agreement. But smart contracts can't access data from outside of their blockchain network. An oracle is a way for a **blockchain or smart contract to interact with external data**.

With blockchains being deterministic one-way streets, an oracle is the path between off-chain and on-chain events. These oracles are services that send and verify real world occurrences and submit this information to smart contracts, triggering state changes on the blockchain.

Inbound oracles bring off-chain or real-world data to the blockchain, whereas their outbound cousins do the opposite: they inform an entity outside the blockchain of an event that occurred on it.

Blockchain Oracles

:::caution Content disclaimer

Please view the third-party content disclaimer here.

:::

To integrate your dApp with Oracles on Polygon, you can choose to go with one of the following solutions:

1. API3
2. Chainlink
3. BandChain
4. Razor
5. Tellor
6. UMA

Resources

1. What is the blockchain oracle problem?
2. What is Blockchain Oracle
3. Types of Blockchain Oracle
4. Tutorial: Getting decentralized price data
5. Tutorial: How to integrate Razor on Polygon

---

id: optimisticoracle title: UMA's Optimistic Oracle sidebar_label: UMA description: UMA's Optimistic Oracle allows contracts to quickly request and receive any kind of data keywords: - wiki - polygon - oracle - UMA - Optimistic Oracle image: https://wiki.polygon.technology/img/polygon-logo.png

:::caution Content disclaimer

Please view the third-party content disclaimer here.

:::

UMA's Optimistic Oracle allows contracts to quickly request and receive any kind of data. UMA's oracle system is comprised of two core components:

1. Optimistic Oracle
2. Data Verification Mechanism (DVM)

Optimistic Oracle

UMA's **Optimistic Oracle** allows contracts to quickly request and receive price information. The Optimistic Oracle acts as a generalized escalation game between contracts that initiate a price request and UMA's dispute resolution system known as the Data Verification Mechanism (DVM).

Prices proposed by the Optimistic Oracle will not be sent to the DVM unless it is disputed. This enables contracts to obtain price information within any pre-defined length of time without writing the price of an asset on-chain.

Data Verification Mechanism (DVM)

If a dispute is raised, a request is sent to the DVM. All contracts built on UMA use the DVM as a backstop to resolve disputes. Disputes sent to the DVM will be resolved 48 hours after UMA tokenholders vote on the price of the asset at a given time. Contracts on UMA do not need to use the Optimistic Oracle unless it requires a price of an asset faster than 48 hours.

The Data Verification Mechanism (DVM) is the dispute resolution service for contracts built on UMA Protocol. The DVM is powerful because it encompasses an element of human judgment to ensure contracts are securely and correctly managed when issues arise from volatile (and sometimes manipulatable) markets.

Optimistic Oracle Interface

The Optimistic Oracle is used by financial contracts or any third party to retrieve prices. Once a price is requested, anyone can propose a price in response. Once proposed, the price goes through a liveness period where anyone can dispute the proposed price and send the disputed price to the UMA DVM for settlement.

:::info

This section explains how different participants can interact with the Optimistic Oracle. To view the most updated mainnet, kovan or L2 deployments of the Optimistic Oracle contracts, refer to the production addresses.

:::

There are twelve methods that make up the Optimistic Oracle interface.

- `requestPrice`
- `proposePrice`
- `disputePrice`
- `settle`
- `hasPrice`
- `getRequest`
- `settleAndGetPrice`
- `setBond`
- `setCustomLiveness`
- `setRefundOnDispute`
- `proposePriceFor`
- `disputePriceFor`

requestPrice

Requests a new price. This must be for a registered price identifier. Note that this is called automatically by most financial contracts that are registered in the UMA system, but can be called by anyone for any registered price identifier. For example, the Expiring Multiparty (EMP) contract calls this method when its `expire` method is called.

Parameters:

- `identifier` : price identifier being requested.
- `timestamp` : timestamp of the price being requested.
- `ancillaryData` : ancillary data representing additional args being passed with the price request.
- `currency` : ERC20 token used for payment of rewards and fees. Must be approved for use with the DVM.
- `reward` : reward offered to a successful proposer. Will be paid by the caller. Note: this can be 0.

proposePrice

Proposes a price value for an existing price request.

Parameters:

- `requester` : sender of the initial price request.
- `identifier` : price identifier to identify the existing request.
- `timestamp` : timestamp to identify the existing request.
- `ancillaryData` : ancillary data of the price being requested.
- `proposedPrice` : price being proposed.

disputePrice

Disputes a price value for an existing price request with an active proposal.

Parameters:

- `requester` : sender of the initial price request.

- `identifier` : price identifier to identify the existing request.
- `timestamp` : timestamp to identify the existing request.
- `ancillaryData` : ancillary data of the price being requested.

settle

Attempts to settle an outstanding price request. Will revert if it can't be settled.

Parameters:

- `requester` : sender of the initial price request.
- `identifier` : price identifier to identify the existing request.
- `timestamp` : timestamp to identify the existing request.
- `ancillaryData` : ancillary data of the price being requested.

hasPrice

Checks if a given request has resolved or been settled (i.e the optimistic oracle has a price).

Parameters:

- `requester` : sender of the initial price request.
- `identifier` : price identifier to identify the existing request.
- `timestamp` : timestamp to identify the existing request.
- `ancillaryData` : ancillary data of the price being requested.

getRequest

Gets the current data structure containing all information about a price request.

Parameters:

- `requester` : sender of the initial price request.
- `identifier` : price identifier to identify the existing request.
- `timestamp` : timestamp to identify the existing request.
- `ancillaryData` : ancillary data of the price being requested.

settleAndGetPrice

Retrieves a price that was previously requested by a caller. Reverts if the request is not settled or settleable. Note: this method is not view so that this call may actually settle the price request if it hasn't been settled.

Parameters:

- `identifier` : price identifier to identify the existing request.
- `timestamp` : timestamp to identify the existing request.
- `ancillaryData` : ancillary data of the price being requested.

setBond

Set the proposal bond associated with a price request.

Parameters:

- `identifier` : price identifier to identify the existing request.
- `timestamp` : timestamp to identify the existing request.
- `ancillaryData` : ancillary data of the price being requested.
- `bond` : custom bond amount to set.

setCustomLiveness

Sets a custom liveness value for the request. Liveness is the amount of time a proposal must wait before being auto-resolved.

Parameters:

- `identifier` : price identifier to identify the existing request.
- `timestamp` : timestamp to identify the existing request.
- `ancillaryData` : ancillary data of the price being requested.
- `customLiveness` : new custom liveness.

setRefundOnDispute

Sets the request to refund the reward if the proposal is disputed. This can help to "hedge" the caller in the event of a dispute-caused delay. Note: in the event of a dispute, the winner still receives the other's bond, so there is still profit to be made even if the reward is refunded.

Parameters:

- `identifier` : price identifier to identify the existing request.
- `timestamp` : timestamp to identify the existing request.
- `ancillaryData` : ancillary data of the price being requested.

disputePriceFor

Disputes a price request with an active proposal on another address' behalf. Note: this address will receive any rewards that come from this dispute. However, any bonds are pulled from the caller.

Parameters:

- `disputer` : address to set as the disputer.
- `requester` : sender of the initial price request.
- `identifier` : price identifier to identify the existing request.
- `timestamp` : timestamp to identify the existing request.
- `ancillaryData` : ancillary data of the price being requested.

proposePriceFor

Proposes a price value on another address' behalf. Note: this address will receive any rewards that come from this proposal. However, any bonds are pulled from the caller.

Parameters:

- `proposer` : address to set as the proposer.
- `requester` : sender of the initial price request.
- `identifier` : price identifier to identify the existing request.
- `timestamp` : timestamp to identify the existing request.
- `ancillaryData` : ancillary data of the price being requested.
- `proposedPrice` : price being proposed.

Integrating the Optimistic Oracle

This demo will set up an `OptimisticDepositBox` contract which custodies a user's ERC-20 token balance.

On a local testnet blockchain, the user will deposit wETH (Wrapped Ether) into the contract and withdraw wETH denominated in USD. For example, if the user wants to withdraw $10,000 USD of wETH, and the ETH/USD exchange rate is $2,000, they would withdraw 5 wETH.

- The user links the `OptimisticDepositBox` with one of the price identifiers enabled on the DVM.
- The user deposits wETH into the `OptimisticDepositBox` and register it with the `ETH/USD` price identifier.
- The user can now withdraw a USD-denominated amount of wETH from their `DepositBox` via smart contract calls, with the Optimistic Oracle enabling optimistic on-chain pricing.

In this example, the user would not have been able to transfer USD-denominated amounts of wETH without referencing an off-chain `ETH/USD` price feed. The Optimistic Oracle therefore enables the user to "pull" a reference price.

Unlike price requests to the DVM, a price request to the Optimistic Oracle can be resolved within a specified liveness window if there are no disputes, which can be significantly shorter than the DVM voting period. The liveness window is configurable, but is typically two hours, compared to 2-3 days for settlement via the DVM.

The price requestor is not currently required to pay fees to the DVM. The requestor can offer a reward for the proposer who responds to a price request, but the reward value is set to `0` in this example.

The price proposer posts a bond along with their price, which will be refunded if the price is not disputed, or if a dispute is resolved in the proposer's favor. Otherwise, this bond is used to pay the final fee to the DVM and pay a reward to a successful disputer.

In the demo, the requestor does not require an additional bond from the price proposer, so the total bond posted is equal to the wETH final fee currently 0.2 wETH. See the `proposePriceFor` function in the `OptimisticOracle` contract for implementation details.

Running the Demo

1. Ensure that you have followed all the prerequisite setup steps here.
2. Run a local Ganache instance (i.e. not Kovan/Ropsten/Rinkeby/Mainnet) with `yarn ganache-cli --port 9545`
3. In another window, migrate the contracts by running the following command:

```
yarn truffle migrate --reset --network test
```

1. To deploy the `OptimisticDepositBox` contract and go through a simple user flow, run the following demo script from the root of the repo:

```
yarn truffle exec ./packages/core/scripts/demo/OptimisticDepositBox.js --network test
```

You should see the following output:

```
1. Deploying new OptimisticDepositBox
  - Using wETH as collateral token
  - Pricefeed identifier for ETH/USD is whitelisted
  - Collateral address for wETH is whitelisted
  - Deployed an OptimisticOracle
  - Deployed a new OptimisticDepositBox


2. Minting ERC20 to user and giving OptimisticDepositBox allowance to transfer collateral
  - Converted 10 ETH into wETH
  - User's wETH balance: 10
  - Increased OptimisticDepositBox allowance to spend wETH
  - Contract's wETH allowance: 10
```

```
3. Depositing ERC20 into the OptimisticDepositBox
  - Deposited 10 wETH into the OptimisticDepositBox
  - User's deposit balance: 10
  - Total deposit balance: 10
  - User's wETH balance: 0


4. Withdrawing ERC20 from OptimisticDepositBox
  - Submitted a withdrawal request for 10000 USD of wETH
  - Proposed a price of 2000000000000000000000 ETH/USD
  - Fast-forwarded the Optimistic Oracle and Optimistic Deposit Box to after the liveness window so we can settle.
  - New OO time is [fast-forwarded timestamp]
  - New ODB time is [fast-forwarded timestamp]
  - Executed withdrawal. This also settles and gets the resolved price within the withdrawal function.
  - User's deposit balance: 5
  - Total deposit balance: 5
  - User's wETH balance: 5
```

Explaining the Contract Functions

The `OptimisticDepositBox` contract code shows how to interact with the Oracle.

The `constructor` function includes a `_finderAddress` argument for the UMA `Finder` contract, which maintains a registry of the `OptimisticOracle` address, approved collateral and price identifier whitelists, and other important contract addresses.

This allows the `constructor` to check that the collateral type and price identifier are valid, and allows the `OptimisticDepositBox` to find and interact with the `OptimisticOracle` later.

The `requestWithdrawal` function includes an internal call to the `OptimisticOracle` requesting the `ETH/USD` price. Once it's returned, the user can call `executeWithdrawal` to complete the withdrawal.

There is much more information and explanation in the code comments, so please take a look if you're interested in learning more.

Additional Resources

Here are some additional resources regarding the UMA DVM:

- Technical Architecture
- Economic Architecture
- Blog post on UMA's DVM design
- Whitepaper on UMA's DVM design
- Research repo for optimal fee policy
- UMIP repo for governance proposals

---

id: razor title: Razor sidebar_label: Razor description: Razor Network is a decentralized oracle providing data to the smart contracts using a network of stakers keywords: - wiki - polygon - razor - oracle image: https://wiki.polygon.technology/img/polygon-logo.png

:::caution Content disclaimer

Please view the third-party content disclaimer here.

:::

Razor Network is a decentralized oracle providing data to the smart contracts using a network of stakers. The network consists of validators who lock in their tokens as a **Stake** and provide data to the network. The honest validators are rewarded and those who report incoherently are penalized.

The core of Razor Network is a set of smart contracts that can run on any Ethereum-compatible blockchain. Razor relies on the underlying blockchain for providing certain properties such as censorship resistance, security from network partition attacks, etc.

Usage

This oracle network is currently running on the Polygon Mumbai testnet. To get started with connecting to the data feed, check out this **tutorial**.

The Razor Network can be explored using Razorscan.

---

title: Tellor description: "A guide to integrate the Tellor oracle into your Polygon contract." author: "Tellor" lang: en sidebar: true tags: ["solidity", "smart contracts", "price feeds", "oracles", "Polygon", "Matic", "Tellor"] skill: beginner published: 2022-02-10 source: Tellor Docs sourceUrl: https://docs.tellor.io/tellor/

:::caution Content disclaimer

Please view the third-party content disclaimer here.

:::

Tellor is an oracle that provides censorship resistant data that is secured by simple crypto-economic incentives. Data can be provided by anyone and checked by everyone. Tellor's flexible structure can provide any data at any time interval to allow for easy experimentation/innovation.

(Soft) Prerequisites

We're assuming the following about your coding skill-level to focus on the oracle aspect.

Assumptions:

- you can navigate a terminal
- you have npm installed
- you know how to use npm to manage dependencies

Tellor is a live and open-sourced oracle ready for implementation. This beginner's guide is here to showcase the ease with which one can get up and running with Tellor, providing your project with a fully decentralized and censorship-resistant oracle.

Overview

Tellor is an oracle system where parties can request the value of an off-chain data point (e.g. BTC/USD) and reporters compete to add this value to an on-chain data-bank, accessible by all Polygon smart contracts. The inputs to this data-bank are secured by a network of staked reporters. Tellor utilizes crypto-economic incentive mechanisms. Honest data submissions by reporters are rewarded by the issuance of Tellor's token. Any bad actors are quickly punished and removed from the network by a dispute mechanism.

In this tutorial we'll go over:

- Setting up the initial toolkit you'll need to get up and running.
- Walk through a simple example.
- List out testnet addresses of networks you currently can test Tellor on.

UsingTellor

The first thing you'll want to do is install the basic tools necessary for using Tellor as your oracle. Use this package to install the Tellor User Contracts:

```
npm install usingtellor
```

Once installed this will allow your contracts to inherit the functions from the contract 'UsingTellor'.

Great! Now that you've got the tools ready, let's go through a simple exercise where we retrieve the bitcoin price:

BTC/USD Example

Inherit the UsingTellor contract, passing the Tellor address as a constructor argument:

Here's an example:

```
import "usingtellor/contracts/UsingTellor.sol";

contract PriceContract is UsingTellor {

  uint256 public btcPrice;

  //This Contract now has access to all functions in UsingTellor

  constructor(address payable _tellorAddress) UsingTellor(_tellorAddress) public {}

  function setBtcPrice() public {

    bytes memory _b = abi.encode("SpotPrice",abi.encode("btc","usd"));
    bytes32 _queryID = keccak256(_b);

    uint256 _timestamp;
    bytes _value;

    (_value, _timestamp) = getDataBefore(_queryId, block.timestamp - 15 minutes);

    btcPrice = abi.decode(_value,(uint256));
  }
}
```

Addresses:

Tellor Tributes: `0xe3322702bedaaed36cddab233360b939775ae5f1`

Oracle: `0xD9157453E2668B2fc45b7A803D3FEF3642430cC0`

Looking to do some testing first?:

Polygon Mumbai Testnet: `0xD9157453E2668B2fc45b7A803D3FEF3642430cC0`

Test Tributes: `0xCE4e32fE9D894f8185271Aa990D2dB425DF3E6bE`

Need some test tokens? Tweet us at '@trbfaucet'

For ease of use, the UsingTellor repo comes with a version of the Tellor Playground contract for easier integration. See here for a list of helpful functions.

For a more robust implementation of the Tellor oracle, check out the full list of available functions here.

Still have questions? Join the community here!

---

id: umb title: Umbrella Network sidebar_label: Umbrella Network description: Umbrella Network is a decentralized oracle service that provides blockchain projects with secure, scalable, and customizable data solutions. keywords: - oracle - umbrella - network - Layer 2 - offchain - onchain - data - prices image: https://wiki.polygon.technology/img/polygon-logo.png

import useBaseUrl from '@docusaurus/useBaseUrl';

:::caution Content disclaimer

Please view the third-party content disclaimer here.

:::

Umbrella Network is a decentralized oracle service that provides blockchain projects with secure, scalable, and customizable data solutions.

The service leverages a network of decentralized community validators to ensure the reliability of the oracle data.

Solutions

Umbrella Network provides three data consumption options: on-chain data, layer 2 data, and on-demand data.

On-chain data

Retrieve data feeds directly on-chain. Projects can define parameters such as deviation triggers and heartbeat to manage the frequency and precision of the updates. This is the simplest and most straightforward approach for reading data.

Layer-2 data

Read data feeds off-chain and verify them on-chain to ensure secure utilization within your dApp. Prices are regularly updated off-chain at a predefined frequency, and web3 apps fetch this data when needed and submit it on-chain. This approach is particularly advantageous when your project necessitates handling large volumes of data, as storing it entirely on-chain would be cost-prohibitive.

On-demand data

On-demand data fetching planned for Q3 2023.

Resources

Technical documentation

The official technical documentation can be found here.

Custom feeds

In case your project requires a custom feed, please contact the Umbrella team here.

id: crusthelpers title: Crust Network description: Decentralized IPFS storage network keywords: - Crust Network - crustfiles - crust-polygon bridge - IPFS storage - matic - polygon - docs image: https://wiki.polygon.technology/img/polygon-logo.png

:::caution Content disclaimer

Please view the third-party content disclaimer here.

:::

Crust Network provides a decentralized storage network for Web3. It is designed to realize the core values of decentralization, privacy and assurance. Crust supports multiple storage-layer protocols such as IPFS, and exposes instantly accessible on-chain storage functions to users.

The protocol matches people who have hard drive space to spare with those who need to store data or host content. Crust supports most smart contract platforms with its cross-chain solution, including Polygon. Find out all the platforms that Crust storage supports on their official website.

:::info Learn more about Crust Network

Check out the **Decentralized Storage Market** and **Cross-chain Storage Solution** offered by Crust Network. Also, you can start building with **Crust Build-101**.

:::

Crust Storage Helpers

- Crust Files: Built by the Crust community, Crust Files is the first personal cloud storage for Web3 that supports multi-web3 wallet login and provides fully decentralized and privacy storage function for all users.

- IPFS W3Auth Gateway: A Web3 authenticated IPFS gateway that supports all the IPFS *WRITE* functions and helps users upload files or host content after passing the W3Auth verification.
- IPFS W3Auth Pinning Service: A standard IPFS pinning service with Web3 authentication that allows users to access Crust storage resources without needing to interact with the Crust blockchain.
- Crust Github Action: A standard Github action which can include fully decentralized CI/CD hosting with Crust.

---

id: filecoinhelpers title: Filecoin description: Storage Helpers by IPFS and Filecoin keywords: - Filecoin - nft.storage - web3.storage - esturay - filecoin-polygon bridge - IPFS - docs - polygon - matic - lighthouse image: https://wiki.polygon.technology/img/polygon-logo.png

:::caution Content disclaimer

Please view the third-party content disclaimer here.

:::

Filecoin is built on top of IPFS and supports storing data long-term via on-chain deals. Together, they help us break free from centralized services while conveniently allowing us to enjoy the same luxuries of speed and guaranteed storage that centralized services would bring.

Storage Helpers (IPFS + Filecoin)

- Estuary: Pinning service that stores and retrieves data on both IPFS and Filecoin networks via simple API calls (video)
- Lighthouse.storage: Permanent storage on IPFS and Filecoin that allows users to pay once and store forever. Lighthouse can also be used to store private data and build token gated applications on Filecoin.
- NFT.storage: NFT storage service that stores and retrieves data relating to NFTs on IPFS and Filecoin (video)
- Web3.storage: Data storage service that stores and retrieves data on IPFS and Filecoin (video)
- Textile Powergate: Highly configurable wrapper for IPFS+Filecoin inside a Docker container, with bridges Polygon, and other smart contract networks
- Fleek Space Daemon and Space SDK: For decentralized browser, mobile, or desktop development

---

id: ipfs title: IPFS description: "IPFS - distributed system for storing and accessing data." keywords: - IPFS - matic - docs - polygon - storage - blockchain image: https://wiki.polygon.technology/img/polygon-logo.png

Context

Polygon blockchain reduces transaction costs to store data versus Ethereum mainnet; however, even these lower costs add up quickly when storing sizable files. Developers also must consider block size constraints and transaction speed limitations when storing data onchain. One solution which addresses all of these concerns is IPFS, the InterPlanetary File System.

What is IPFS?

IPFS is a distributed system for storing and accessing files, websites, applications, and data. IPFS uses decentralization, content addressing, and a robust peer-to-peer network of active participants to allow users to store, request, and transfer verifiable data with each other.

Decentralization makes it possible to download a file from many locations that aren't managed by one organization, providing resilience and censorship resistance right out of the box.

Content addressing uses cryptography to create a uniquely verifable hash based upon what is in a file rather than where it is located. The resulting content identifier (CID) provides assurance a piece of data is identical regardless of where it is stored.

Finally, an ever growing active community of users makes this peer-to-peer sharing of content possible. Developers upload and pin content to IPFS while Filecoin or Crust storage providers help to ensure persistent storage of that content.

IPFS based storage allows you to simply store the CID for your content rather than loading entire files to Polygon blockchain; allowing for decreased costs, larger file sizes, and provably persistent storage. For more details refer IPFS Docs.

Example Projects

1. Tutorial in scaffold-eth that demonstrates how to mint an NFT on Polygon with IPFS - link
2. Building a full stack web3 app with Next.js, Polygon, Solidity, The Graph, IPFS, and Hardhat - link

---

id: nftstorage title: Mint NFTs description: Mint with NFT.storage and Polygon. keywords: - nft.storage - filecoin - matic - polygon - docs - mint nfts image: https://wiki.polygon.technology/img/polygon-logo.png

:::caution Content disclaimer

Please view the third-party content disclaimer here.

:::

This tutorial will teach you to mint an NFT using the Polygon blockchain and IPFS/Filecoin storage via NFT.Storage. Polygon, a Layer 2 scaling solution for Ethereum, is often chosen by developers for its speed and lower transaction costs while maintaining full compatibility with Ethereum's EVM. The tutorial will walk

you through the creation and deployment of a standardized smart contract, storing metadata and assets on IPFS and Filecoin via the NFT.Storage API and minting the NFT to your own wallet on Polygon.

Introduction

In this tutorial we will aim to fulfill three characteristics with our minting process:

1. *Scalability* of the minting process in terms of cost and throughput. If the use case aims to rapidly create NFTs, the underlying technology needs to handle all minting requests and the minting should be cheap.
2. *Durability* of the NFT, as assets can be long-lived and therefore need to remain usable during their full lifetime.
3. *Immutability* of the NFT and the asset it represents to prevent unwanted changes and malicious actors from changing the digital asset the NFT represents.

Polygon addresses the *scalability* characteristic with their protocol and framework. They are also compatible with Ethereum and its virtual machine, enabling developers to move their code freely between the two blockchains. Likewise, NFT.Storage guarantees *durability* with the power of the underlying Filecoin network and *immutability* by using IPFS' content addressing.

In this tutorial you will get an overview of the NFT minting process, learn how to store a digital asset with NFT.Storage and use this digital asset to mint your NFT on Polygon.

Prerequisites

General knowledge about NFTs will give you background and context. NFT School covers NFT basics, advanced topics and has more tutorials.

To test and run the code found in this tutorial, you will need a working Node.js installation.

You'll also need a Polygon wallet on the Mumbai testnet with a small amount of the MATIC token. Follow the instructions below to get started:

1. **Download and install Metamask**. MetaMask is a crypto wallet and gateway to blockchain apps. It's very easy to use and simplifies a lot of steps, e.g., setting up a Polygon wallet.
2. **Connect MetaMask to Polygon's Mumbai testnet** and select it in the dropdown menu. We will use Polygon's testnet to mint our NFT as it's free of charge.
3. **Receive MATIC token** to your wallet by using the faucet. Select the Mumbai testnet and paste your wallet address from MetaMask into the form. To mint an NFT, we need to pay a small amount of MATIC, which is a fee charged by miners for operations to add new transactions to the blockchain, e.g., minting an NFT or creating a new smart contract.
4. **Copy your private key** from MetaMask by clicking on the three dots in the top right corner and selecting 'Account details'. On the bottom you can find a button to export your private key. Click it and enter your password when prompted. You can copy and paste the private key in a text file for now. We will use it later in the tutorial when interacting with the blockchain.

Lastly, you will need a text or code editor. For more convenience, choose an editor with language support for both JavaScript and Solidity. A good option is Visual Studio Code with the solidity extension enabled.

Preparation

Get an API key for NFT.storage

In order to use NFT.Storage you need an API key. First, head over to NFT.Storage to log in with your email address. You will receive an email with a magic link that signs you in -- no password needed. After you successfully logged in, go to API Keys via the navigation bar. You will find a button to create a **New Key**. When prompted for an API key name, you can freely choose one or use “polygon + NFT.Storage”. You can copy the content of the key column now or reference back to NFT.Storage later in the tutorial.

Set up your workspace

Create a new empty folder that we can use as our workspace for this tutorial. Feel free to choose any name and location on your file system. Open up a terminal and navigate to the newly created folder.

Next, we will install the following Node.js dependencies:

- **Hardhat and Hardhat-Ethers**, a development environment for Ethereum (and Ethereum compatible blockchains like Polygon).
- **OpenZeppelin**, a collection of smart contracts featuring standardized NFT base contracts.
- **NFT.Storage**, a library to connect to the NFT.Storage API.
- **Dotenv**, a library to handle environment files for configuration (e.g., injecting private keys into the script).

Use the following command to install all dependencies at once:

```
npm install hardhat @openzeppelin/contracts nft.storage dotenv @nomiclabs/hardhat-ethers
```

Hardhat needs to be initialized in the current folder. In order to start the initialization, execute:

```
npx hardhat
```

When prompted, choose **Create an empty hardhat.config.js**. Your console output should look like this:

```
✔ What do you want to do? · Create an empty hardhat.config.js
✨ Config file created ✨
```

We will do some modifications to the hardhat configuration file `hardhat.config.js` to support the Polygon Mumbai test network. Open the `hardhat.config.js` that was created in the last step. Please note that we are loading your Polygon wallet private key from an environment file and that this environment file must be kept safe. You can even use other rpc link, as per requirement.

```
/**
* @type import('hardhat/config').HardhatUserConfig
*/
require("@nomiclabs/hardhat-ethers");
require('dotenv').config();
const { PRIVATE_KEY } = process.env;
module.exports = {
  defaultNetwork: "PolygonMumbai",
  networks: {
    hardhat: {
    },
    PolygonMumbai : {
      url: "https://rpc-mumbai.maticvigil.com",
      accounts: [PRIVATE_KEY]
    }
  },
  solidity: {
    version: "0.8.12",
    settings: {
      optimizer: {
        enabled: true,
        runs: 200
      }
    }
  },
}
```

Create a new file called `.env` which will hold your API key for NFT.Storage and your Polygon wallet private key. The content of the `.env` file should look something like:

```
PRIVATE_KEY="Your Private Key"
NFT_STORAGE_API_KEY="Your Api Key"
```

Replace the placeholders with the API key you created during preparation and your Polygon wallet private key.

To keep our project organized, we'll create three new folders:

1. `contracts` , for the Polygon contracts written in Solidity.
2. `assets` , containing the digital asset we will mint as an NFT.
3. `scripts` , as helpers to drive the preparation and minting process.

Execute the following command:

```
mkdir contracts assets scripts
```

Lastly, we will add an image to the `assets` folder. This image will be our artwork that we will upload to NFT.Storage and mint on Polygon. We will name it `MyExampleNFT.png` for now. If you do not have some nice art ready, you can download a simple pattern.

Minting your NFT

Storing asset data with NFT.Storage

We will use NFT.Storage to store our digital asset and its metadata. NFT.Storage guarantees immutability and durability by uploading your digital asset to Filecoin and IPFS automatically. IPFS and Filecoin operate on content identifiers (CID) for immutable referencing. IPFS will provide fast retrieval with its geo-replicated caching and Filecoin guarantees durability with incentivized storage providers.

Create a script called `store-asset.mjs` below the `scripts` directory. The contents are listed below:

```
import { NFTStorage, File } from "nft.storage"
import fs from 'fs'
import dotenv from 'dotenv'
dotenv.config()

const { NFT_STORAGE_API_KEY } = process.env

async function storeAsset() {
   const client = new NFTStorage({ token: NFT_STORAGE_API_KEY })
   const metadata = await client.store({
       name: 'ExampleNFT',
       description: 'My ExampleNFT is an awesome artwork!',
       image: new File(
           [await fs.promises.readFile('assets/MyExampleNFT.png')],
           'MyExampleNFT.png',
           { type: 'image/png' }
       ),
   })
```

```
    console.log("Metadata stored on Filecoin and IPFS with URL:", metadata.url)
}

storeAsset()
    .then(() => process.exit(0))
    .catch((error) => {
        console.error(error);
        process.exit(1);
    });
```

The main part of the script is the `storeAsset` function. It creates a new client connecting to NFT.Storage using the API key you created earlier. Next we introduce the metadata consisting of name, description, and the image. Note that we are reading the NFT asset directly from the file system from the `assets` directory. At the end of the function we will print the metadata URL as we will use it later when creating the NFT on Polygon.

After setting up the script, you can execute it by running:

```
node scripts/store-asset.mjs
```

Your output should look like the listing below, where `HASH` is the CID to the art you just stored.

```
Metadata stored on Filecoin/IPFS at URL: ipfs://HASH/metadata.json
```

Creating your NFT on Polygon

Create the smart contract for minting

First, we will create a smart contract that will be used to mint the NFT. Since Polygon is compatible with Ethereum, we will write the smart contract in Solidity. Create a new file for our NFT smart contract called `ExampleNFT.sol` inside the `contracts` directory. You can copy the code of the listing below:

```
// Contract based on https://docs.openzeppelin.com/contracts/4.x/erc721
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.12;

import "@openzeppelin/contracts/token/ERC721/ERC721.sol";
import "@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol";
import "@openzeppelin/contracts/utils/Counters.sol";
import "@openzeppelin/contracts/access/Ownable.sol";

contract ExampleNFT is ERC721URIStorage, Ownable {
   using Counters for Counters.Counter;
   Counters.Counter private _tokenIds;

   constructor() ERC721("NFT", "ENFT") {}

   function mintNFT(address recipient, string memory tokenURI)
       public onlyOwner
       returns (uint256)
   {
       _tokenIds.increment();

       uint256 newItemId = _tokenIds.current();
       _mint(recipient, newItemId);
       _setTokenURI(newItemId, tokenURI);

       return newItemId;
   }
}
```

To be a valid NFT, your smart contract must implement all the methods of the ERC-721 standard. We use the implementation of the OpenZeppelin library, which already provides a set of basic functionalities and adheres to the standard.

At the top of our smart contract, we import three OpenZeppelin smart contract classes:

1. `\@openzeppelin/contracts/token/ERC721/ERC721.sol` contains the implementation of the basic methods of the ERC-721 standard, which our NFT smart contract will inherit. We use the `ERC721URIStorage,` which is an extension to store not just the assets but also metadata as a JSON file off-chain. Like the contract, this JSON file needs to adhere to ERC-721.
2. `\@openzeppelin/contracts/utils/Counters.sol` provides counters that can only be incremented or decremented by one. Our smart contract uses a counter to keep track of the total number of NFTs minted and to set the unique ID on our new NFT.
3. `\@openzeppelin/contracts/access/Ownable.sol` sets up access control on our smart contract, so only the owner of the smart contract (you) can mint NFTs.

After our import statements, we have our custom NFT smart contract, which contains a counter, a constructor, and a method to actually mint the NFT. Most of the hard work is done by the base contract inherited from OpenZeppelin, which implements most of the methods we require to create an NFT adhering to the ERC-721 standard.

The counter keeps track of the total number of NFTs minted, which is used in the minting method as a unique identifier for the NFT.

In the constructor, we pass in two string arguments for the name of the smart contract and the symbol (represented in wallets). You can change them to whatever you like.

Finally, we have our method `mintNFT` that allows us to actually mint the NFT. The method is set to `onlyOwner` to make sure it can only be executed by the owner of the smart contract.

`address recipient` specifies the address that will receive the NFT at first.

`string memory tokenURI` is a URL that should resolve to a JSON document that describes the NFT's metadata. In our case it's already stored on NFT.Storage. We can use the returned IPFS link to the metadata JSON file during the execution of the method.

Inside the method, we increment the counter to receive a new unique identifier for our NFT. Then we call the methods provided by the base contract from OpenZeppelin to mint the NFT to the recipient with the newly created identifier and setting the URI of the metadata. The method returns the unique identifier after execution.

Deploy the Smart Contract to Polygon

Now, it's time to deploy our smart contract to Polygon. Create a new file called `deploy-contract.mjs` within the `scripts` directory. Copy the contents of the listing below into that file and save it.

```
async function deployContract() {
 const ExampleNFT = await ethers.getContractFactory("ExampleNFT")
 const exampleNFT = await ExampleNFT.deploy()
 await exampleNFT.deployed()
 // This solves the bug in Mumbai network where the contract address is not the real one
 const txHash = exampleNFT.deployTransaction.hash
 const txReceipt = await ethers.provider.waitForTransaction(txHash)
 const contractAddress = txReceipt.contractAddress
 console.log("Contract deployed to address:", contractAddress)
}

deployContract()
 .then(() => process.exit(0))
 .catch((error) => {
   console.error(error);
   process.exit(1);
 });
```

Deploying the contract is done with the helper functions provided by the hardhat library. First, we get the smart contract we created in the previous step with the provided factory. Then we deploy it by calling the respective method and wait for the deployment to be completed. There are a few more lines below the described code to get the correct address in the testnet environment. Save the `mjs` file.

Execute the script with the following command:

```
npx hardhat run scripts/deploy-contract.mjs --network PolygonMumbai
```

If everything is correct, you will see the following output:

```
Contract deployed to address: 0x{YOUR_CONTRACT_ADDRESS}
```

Note that you will need the printed contract address in the minting step. You can copy and paste it into a separate text file and save it for later. This is necessary so the minting script can call the minting method of that specific contract.

Minting the NFT on Polygon

Minting the NFT is now merely calling the contract we just deployed to Polygon. Create a new file called `mint-nft.mjs` inside the `scripts` directory and copy this code from the listing below:

```
const CONTRACT_ADDRESS = "0x00"
const META_DATA_URL = "ipfs://XX"

async function mintNFT(contractAddress, metaDataURL) {
   const ExampleNFT = await ethers.getContractFactory("ExampleNFT")
   const [owner] = await ethers.getSigners()
   await ExampleNFT.attach(contractAddress).mintNFT(owner.address, metaDataURL)
   console.log("NFT minted to: ", owner.address)
}

mintNFT(CONTRACT_ADDRESS, META_DATA_URL)
   .then(() => process.exit(0))
   .catch((error) => {
       console.error(error);
       process.exit(1);
   });
```

Edit the first two lines to insert your **contract address** from the earlier deployment and the **metadata URL** that was returned when storing the asset with NFT.Storage. The rest of the script sets up the call to your smart contract with you as the to-be owner of the NFT and the pointer to the metadata stored on IPFS.

Next, run the script:

```
npx hardhat run scripts/mint-nft.mjs --network PolygonMumbai
```

You can expect to see the following output:

```
NFT minted to: 0x<YOUR_WALLET_ADDRESS>
```

Looking for the sample code from this tutorial? You can find it in the polygon-nft.storage-demo link Github repo.

Conclusion

In this tutorial, we learned how to mint a NFT end-to-end with Polygon and NFT.Storage. This technology combination results in proper decentralization and guarantees *scalability*, *durability*, and *immutability*.

We deployed a custom smart contract to mint our NFT specific to our needs. For this tutorial, we used a simple example based on the ERC-721 standard. However, you can also define complex logic that governs your NFT life cycle. For more complex use cases, the successor standard ERC-1155 is a good place to start. OpenZeppelin, the library we use in our tutorial offers a contracts wizard that helps create NFT contracts.

Successful minting can be seen as the start of the valuable phase of the NFT. The NFT can then be used to prove ownership and can be transferred to other users. Reasons to transfer a NFT might include a successful sale on one of the NFT marketplaces like OpenSea, or a different type of event such as acquiring an item in a NFT based game. Exploring the rich possibilities for NFTs is definitely an exciting next step.

If you'd like help building your NFT project with NFT.storage, we encourage you to join the `#nft-storage` channel on Discord and Slack.

id: adding-a-custom-token title: Adding a Custom Token sidebar_label: Adding a Custom Token description: Build your next blockchain app on Polygon. keywords: - docs - matic - wiki - polygon - custom token image: https://wiki.polygon.technology/img/polygon-logo.png

import useBaseUrl from '@docusaurus/useBaseUrl';

The **Add Custom Token** feature allows you to add any token explicitly and use it with the Polygon Wallet Suite. You just have to search for the token by its contract address, either root or child:

- The **root** is the token contract on Ethereum
- The **child** is the contract on Polygon

How do I find the token contract?

You can search for the token by its name on either Coingecko or Coinmarketcap where you will be able to see its address on the Ethereum chain (for ERC 20 tokens) and other supported subsequent chains like Polygon. The token address on other chains might not be updated but you can surely use the root address for all purposes.

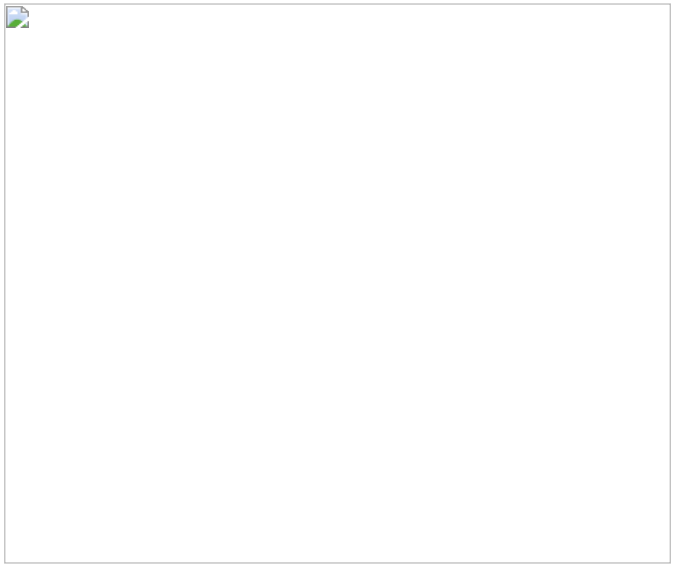
So when selecting a token, you will be able to search by:

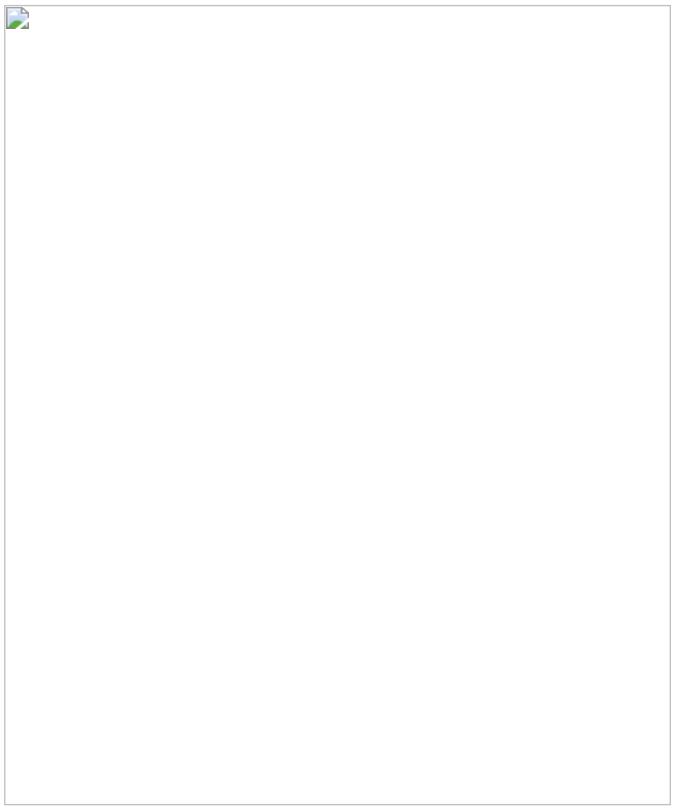
- token symbol
- token name
- contract

Here is how it works:

1. Easily add any token to your list by adding the contract address as a custom token (we support

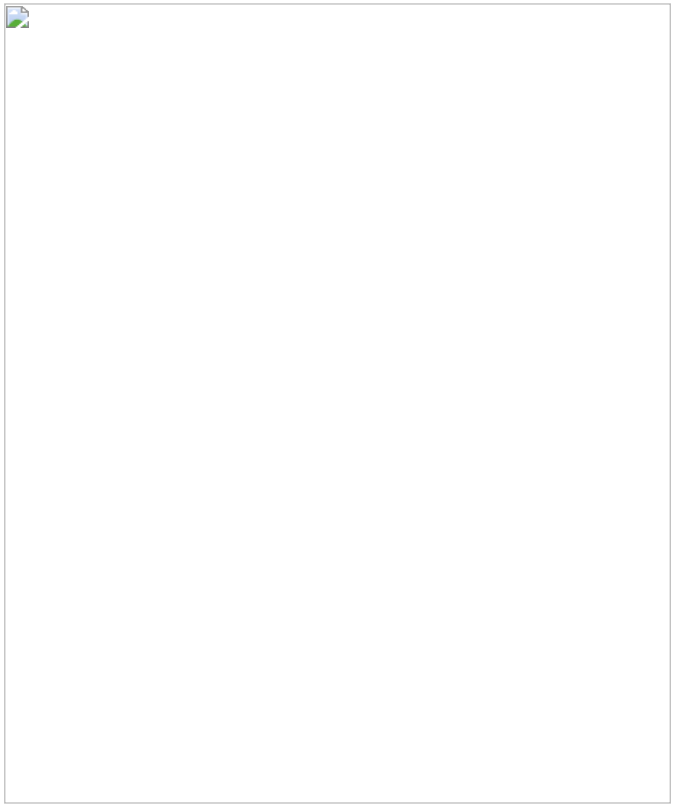
contract addresses on both Polygon or Ethereum):



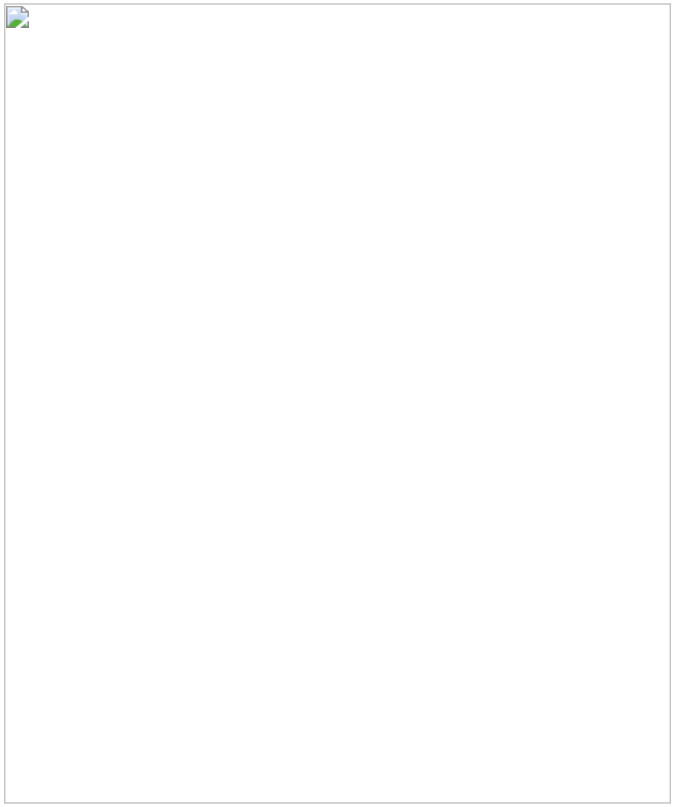


2. Once the token information is fetched, you will see a confirmation screen with all the token information. You can then add it as a custom token which will be stored locally in your system, We suggest you reverify the token contracts twice as there are a lot of clone or scam tokens:

3. Your added token is now shown when selecting a token:



You can also add a token directly from the tokens tab of the **Manage** screen:



--- id: connext title: Crosschain transfers using Connext description: Build your next blockchain app on Polygon. keywords: - docs - matic - connext - polygon image: https://wiki.polygon.technology/img/polygon-logo.png ---

import useBaseUrl from '@docusaurus/useBaseUrl';

:::caution Content disclaimer

Please view the third-party content disclaimer here.

:::

Connext is a crosschain liquidity network that powers fast, fully noncustodial swaps between evm-compatible chains and Ethereum L2 systems.

Ethereum is going multichain. With the growing adoption of evm-compatible chains and L2s, a new challenge has emerged around liquidity fragmentation within the ecosystem. Connext solves this problem by connecting discrete liquidity pools on each chain into a global network, without introducing new, significant trust considerations for users. Developers can leverage this liquidity to build a new class of natively chain-agnostic dApps on Connext.

At a high level, Connext lets users swap assetA on chainA for assetB on chainB using conditional transfers. This happens in a few simple steps:

Alice, a user of Connext, sends a conditional transfer of assetA to Bob. Bob, a liquidity provider (aka a router), sends an equivalent amount of assetB to Alice. Alice unlocks her conditional transfer to receive assetB, which in turn allows Bob to do the same. Routers form the backbone of our network, providing liquidity on different chains and earning fees for doing so. You can learn more about how this works trustlessly in our Protocol Primer.

To setup crosschain transfers from the Ethereum Goerli Testnet to the Polygon Mumbai Testnet in a browser dApp please go through this guide.

id: fortmatic title: Fortmatic description: Use Formatic SDK to integrate your dApp with Polygon keywords: - wiki - polygon - fortmatic - integrate - dapp - sdk image: https://wiki.polygon.technology/img/polygon-logo.png

:::caution Content disclaimer

Please view the third-party content disclaimer here.

:::

Fortmatic SDK allows you to easily integrate your dApp with the Ethereum blockchain, whether you already have a dApp integrated with Web3 or are starting from scratch. Fortmatic provides a smooth and delightful experience for both you and your decentralized application users.

Installation

Use the following command to install Fortmatic's wallet latest version:

```
$ npm i --save fortmatic@latest
```

Example

Here is an example of application using Fortmatic:

```
import Fortmatic from 'fortmatic';
import Web3 from 'web3';

const customNodeOptions = {
    rpcUrl: 'https://rpc-mumbai.matic.today', // your own node url
    chainId: 80001 // chainId of your own node
}

// Setting network to localhost blockchain
const fm = new Fortmatic('YOUR_TEST_API_KEY', customNodeOptions);
window.web3 = new Web3(fm.getProvider());
```---
id: getting-started
title: Wallets
sidebar_label: Getting Started
description: Get a list of supported wallets and manage key strategy.
keywords:
  - wiki
  - polygon
  - wallet
  - integrate
  - guide
image: https://wiki.polygon.technology/img/polygon-logo.png

This document serves as a wallet index for wallets compatible with Polygon.
> Please note that this is not an exhaustive index.

Wallets that support Polygon allow for key management, access to accounts controlled by
private keys, and interfaces that allow users to perform chain actions and sign transactions.

:::info Centralized Exchanges (CEXs)
```
```

For a list of CEXs that support Polygon, visit a third-party tracking website such as [<ins>**CoinMarketCap**</ins>](https://coinmarketcap.com/currencies/polygon/markets).

:::

Native Wallets

The [Polygon Wallet Suite](https://wallet.polygon.technology) is a comprehensive solution for account management on Polygon.

It includes:

- A portfolio view to check balances on Polygon chains
- A bridge for transferring assets between Polygon PoS and Ethereum
- A swap feature for swapping between tokens
- A token mapper for mapping tokens between Polygon and Ethereum
- A faucet for obtaining free test tokens
- A safe bridge for securely bridging between Polygon and Ethereum
- A bridge explorer for checking bridging transactions
- Staking options to secure the Polygon PoS by staking MATIC tokens
- A bridge for transferring assets between Polygon zkEVM and Ethereum
- A zkEVM explorer for block explorer and analytics
- Access to ecosystem dApps

[Polygon Support](https://support.polygon.technology/support/home) can also provide assistance to users and address issues related to the wallet suite.

Interfaces	Custody	Account Type	Multi-Sig	dApp Browser	Platform
[Polygon Wallet Suite](https://wallet.polygon.technology/login/)	non-custodial	EOA	yes (Safe Bridge)	yes	browser
[Hermez Wallet](https://wallet.hermez.io/login)	non-custodial	EOA	no	no	browser

:::tip Stay in the know

Keep up with the latest Wallet Suite updates from the Polygon team and community by subscribing to our [<ins>Notifications</ins>](https://polygon.technology/notifications/).

:::

Partner Wallets

:::caution Third-party wallets

These third-party wallets have integrated Polygon and support a variety of features. **You should do your own due diligence before using them**. The official Polygon Support cannot provide assistance for issues with these wallets or other non-native wallets.

:::

:::caution Content disclaimer

Please view the third-party content disclaimer [<ins>here</ins>](https://github.com/maticnetwork/matic-docs/blob/master/CONTENT_DISCLAIMER.md).

:::

The following wallets are solutions that Polygon Technology has partnered with:

Wallet	Custody	Account Type	Multi-Sig	NFT	dApp Browser	Bridge Support	Fiat On-Ramp	Platforms
[1inch](https://1inch.io/wallet/)	non-custodial	EOA	no	interface	yes	yes	yes	mobile
[Alpha Wallet](https://alphawallet.com/)	non-custodial	EOA	no	interface	yes	yes	yes	mobile, api/sdk
[Atomic Wallet](https://atomicwallet.io/)*	non-custodial	EOA	no	no	no	no	yes	mobile, desktop, api/sdk
[Ambire](https://www.ambire.com/)	non-custodial	smart contract	no	interface	no	yes	yes	browser
[BitKeep](https://bitkeep.com/)	non-custodial	EOA	no	interface	yes	yes	yes	mobile, browser extension
[Bitski](https://www.bitski.com/)	custodial	EOA	no	interface	no	yes	no	browser, api/sdk
[Coin Wallet](https://coin.space/)	non-custodial	EOA	no	no	no	no	yes	mobile,

```
browser, desktop        |
| [Coin98](https://coin98.com/wallet)   | non-custodial         | EOA   | no    | interface     | yes   | yes   | yes   |
mobile, browser, api/sdk        |
| [Coinbase](https://www.coinbase.com/wallet)   | hybrid        | EOA   | no    | interface     | yes   | yes   | yes   |
mobile, browser, api/sdk        |
| [CypherD](https://cypherd.io/)        | non-custodial         | EOA   | no    | yes   | yes   | yes   |  yes  | mobile
|
| [D'Cent](https://dcentwallet.com/)    | hybrid        | EOA   | no    | interface     | yes   | yes   | no    | mobile
|
| [Exodus](https://www.exodus.com/)     | non-custodial         | EOA   | no    | yes   | no    |  no   | yes   | mobile,
desktop         |
| [Gnosis Safe](https://gnosis-safe.io/)        | non-custodial         | smart contract        | yes   | interface     | no
| no    | no    | mobile, browser, desktop, api/sdk     |
| [Guarda](https://guarda.com/)         | non-custodial         | EOA   | no    | no    | no    | yes   | yes   | mobile,
browser, desktop        |
| [Huobi](https://www.itoken.com/en)    | non-custodial         | EOA   | no    | yes   | yes   | yes   | no    | mobile
|
| [Ledger](https://www.ledger.com/)     | non-custodial         | EOA   | no    | interface     | no    | no    | yes   |
hardware, mobile, desktop       |
| [Loopring](https://loopring.io/#/)    | non-custodial         | smart contract        | no    | no    | no    | no | no
| mobile, api/sdk       |
| [Magic](https://fortmatic.com/)*      | custodial     | EOA   | no    | no    | no    |       |       | mobile, browser,
api/sdk         |
| [MathWallet](https://mathwallet.org/en-us/)   | custodial     | EOA   | no    | no    | no    | yes   | yes   | mobile,
browser, api/sdk        |
| [MetaMask](https://metamask.io/)*     | non-custodial         | EOA   | no    | interface     | yes   | no    | no    |
mobile, browser, api/sdk        |
| [Multis](https://multis.co/)*         | non-custodial         | EOA   | no    | no    | no    |       |  yes  | mobile,
desktop         |
| [MyEtherWallet](https://www.myetherwallet.com/)*      | non-custodial         | EOA   | no    | interface     | no    |
| yes | mobile  |
| [Omni](https://omni.app/)     | non-custodial         | EOA   | no    | interface     | no    | yes|          | mobile,
api/sdk         |
| [Opera Crypto Browser](https://www.opera.com/crypto/next)*    | non-custodial         | EOA   | no    | support       | yes
|       |       | mobile, browser       |
| [Paraswap Multichain Wallet](https://apps.apple.com/dz/app/paraswap-multichain-wallet/id1584610690)*  | non-custodial
| EOA   | no    | support       | yes   |       |       | Apple iOS     |
| [Pillar](https://www.pillar.fi/)      | non-custodial         | EOA   | no    | interface     | no    |       | yes | mobile
|
| [Rainbow](https://rainbow.me/)        | non-custodial         | EOA   | no    | interface     | yes   |       | no | mobile,
api/sdk         |
| [SafePal](https://safepal.io/)        | non-custodial         | EOA   | no    | no    | yes   | yes |         | hardware,
mobile, api/sdk         |
| [Sequence](https://sequence.app/auth)         | non-custodial         | smart contract        | no    | interface     | no
|       |       | browser, api/sdk      |
| [SimpleHold](https://simplehold.io/)  | non-custodial         | EOA   | yes   | no    | no    |       | yes | mobile, api/sdk
|
| [TokenPocket](https://www.tokenpocket.pro/en)         | non-custodial         | EOA   | no    | support       | yes   | yes
| yes   | mobile, browser, api/sdk      |
| [Torus](https://toruswallet.io/)      | non-custodial         | EOA   | yes   | support       | no    | no    | no    |
browser, api/sdk        |
| Trezor| non-custodial         | EOA   | no    | support       | no    |       |       | hardware, mobile      |
| [Trust Wallet](https://trustwallet.com/)      | non-custodial         | EOA   | no    | support       | yes   |       |yes|
mobile  |
| [Unstoppable](https://unstoppable.money/)     | non-custodial         | EOA   | no    | yes   | yes   |       |no| mobile,
api/sdk         |
| [Venly](https://www.venly.io/)        | hybrid        | smart contract        | no    | interface     | no    |       |
| browser, api/sdk      |
| [Wirex](https://wirexapp.com/en/wirex-wallet)*        | non-custodial         | EOA   | yes   | no    | no    |       |
| mobile        |
| [XDeFi](https://www.xdefi.io/)        | non-custodial         | EOA   | no    | interface     | no    | no    | no    |
browser         |
| [Zerion](https://zerion.io/)  | non-custodial         | EOA   | no    | yes   | yes   |yes|   | mobile, browser       |

:::info Non-native wallet support

Wallets denoted with * in the table above are not natively supported with the wallet software
and require manual steps to add the Polygon network.

:::

## Key Management Strategy

The following basic steps allow for the integration of a client-side application with Polygon:

1. **Set up Web3**: [web3.js](https://web3js.readthedocs.io/) is a javascript library that
   allows a client-side application to talk to the blockchain. We configure web3.js to communicate
   via a developer-based wallet like MetaMask. Use the [web3.js docs](https://web3js.readthedocs.io/en/v1.2.2/getting-
```

started.html#adding-web3-js) to learn about adding `web3.js` to your project.
2. **Set up an Account**: You will be able to send transactions (specifically those that alter the
   state of the blockchain).
3. **Instantiate Contracts**: Once a web3 object is in place, we next instantiate our deployed contract,
   with which we interact.
4. **Call functions**: Fetch data via functions in the contract through our contract object.
---
id: config-polygon-on-metamask
title: Add Polygon Network
description: Integrate Polygon with MetaMask by using Polygonscan or adding it manually.
keywords:
  - wiki
  - polygon
  - metamask
  - polygonscan
image: https://wiki.polygon.technology/img/polygon-logo.png
---

import useBaseUrl from '@docusaurus/useBaseUrl';
import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';

:::caution Content disclaimer

Please view the third-party content disclaimer [<ins>here</ins>](https://github.com/maticnetwork/matic-docs/blob/master/CONTENT_DISCLAIMER.md).

:::

In order to view the flow of funds in your accounts, on the Polygon Network, you will need to configure Polygon testnet and mainnet URLs on Metamask.

There are two ways to do it:
1. [Using Polygonscan](/tools/wallets/metamask/config-polygon-on-metamask.md#polygon-scan)
2. [Add the Polygon network manually](/tools/wallets/metamask/config-polygon-on-metamask.md#add-the-polygon-network-manually)

:::tip
Find a tip at the end of this article on how to easily add **Polygon Mainnet** to Metamask.
:::

Using Polygonscan

:::note
Please make sure you have already installed <ins>**[Metamask](https://metamask.io/)**</ins>!
:::

<Tabs
  defaultValue="mainnet"
  values={[
    { label: 'PoS Mainnet', value: 'mainnet', },
    { label: 'Mumbai Testnet', value: 'mumbai', },
  ]
}>

<TabItem value="mumbai">

Please follow the steps to add Polygon's Mumbai-Testnet:

1. Navigate to mumbai.polygonscan.com
2. Scroll down to the bottom of the page and click on the button `Add Mumbai Network`

<img src={useBaseUrl("img/metamask/testnet-button.png")} />

3. Once you click the button you will see a MetaMask Notification, now click on **Approve**.

  <img src={useBaseUrl("img/metamask/develop/testnet-addnetwork.png")} width="357" height="600" />

</br>

You will be directly switched to Polygon’s Mainnet now in the network dropdown list. You can now close the dialog.

</TabItem>

```
<TabItem value="mainnet">

Please follow the steps to add Polygon’s Mainnet:

1. Navigate to [polygonscan.com](https://polygonscan.com/)
2. Scroll down to the bottom of the page and click on the button `Add Polygon Network`

<img src={useBaseUrl("img/metamask/mainnet-button.png")} />
<p></p>
<br></br>
<br></br>

3. Once you click the button you will see a MetaMask Notification, now click on **Approve**.
You will be directly switched to Polygon’s Mainnet now in the network dropdown list. You can now close the dialog.

<div align="center">
<img src={useBaseUrl("img/metamask/mainnet-addnetwork.png")} width="357" height="600" />
</div>


</TabItem>

</Tabs>

If you are facing any issue, **add the network manually** according to the steps given below.

### Add the Polygon network manually

<Tabs
  defaultValue="mainnet"
  values={[
    { label: 'PoS Mainnet', value: 'mainnet', },
    { label: 'Mumbai Testnet', value: 'mumbai', },
  ]
}>

<TabItem value="mumbai">
To add Polygon's Mumbai-Testnet, click on the Network selection dropdown and then click on Custom RPC.

<img src={useBaseUrl("img/metamask/select-network.png")} />
<br></br>

It will open up a form with 2 tabs on the top, Settings and Info. In the Settings tab you can add `Matic Mumbai` in the Network Name field, URL `https://rpc-mumbai.maticvigil.com/` in the New RPC URL field, `80001` in Chain ID field, `MATIC` in Currency Symbol field and `https://mumbai.polygonscan.com/` in Block Explorer URL field.
<br></br>

<img src={useBaseUrl("img/metamask/metamask-settings-mumbai.png")} />
<br></br>

Once you’ve added the URL in the New Network field, click on Save. You will be directly switched to Polygon’s Mumbai-Testnet now in the network dropdown list. You can now close the dialog.
</TabItem>

<TabItem value="mainnet">
To add Polygon’s Mainnet, click on the Network selection dropdown and then click on Custom RPC.

<img src={useBaseUrl("img/metamask/select-network.png")} />

It will open up a form with 2 tabs on the top, Settings and Info. In the Settings tab you can add `Polygon Mainnet` in the Network Name field, URL `https://polygon-rpc.com/` in the New RPC URL field, `137` in Chain ID field, `MATIC` in Currency Symbol field and `https://polygonscan.com/` in Block Explorer URL field.

<img src={useBaseUrl("img/metamask/metamask-settings-mainnet.png")} />
<br></br>
<br></br>

Once you’ve added the information click on Save. You will be directly switched to Polygon’s Mainnet now in the network dropdown list. You can now close the dialog.
</TabItem>
</Tabs>

**You have successfully added Polygon Network to your Metamask!**

:::tip Easy way to add Polygon mainnet
If you just need to add Polygon Mainnet, you can follow these steps:

1. On Metamask, select the Network tab:
```

```
<div align="center">
<img src={useBaseUrl("img/metamask/add-polygon-mainnet.png")} width="400"/>
</div>

2. Click on **Add Network**:
<div align="center">
<img src={useBaseUrl("img/metamask/add-polygon-mainnet-2.png")} width="400" />
</div>

3.  Find Polygon Mainnet on the list of available networks:
<div align="center">
<img src={useBaseUrl("img/metamask/add-polygon-mainnet-3.png")} width="800" />
</div>

4. Approve the network addition:
<div align="center">
<img src={useBaseUrl("img/metamask/add-polygon-mainnet-4.png")} width="400" />
</div>

Now you may see the Polygon Mainnet on the Networks tab:
<div align="center">
<img src={useBaseUrl("img/metamask/add-polygon-mainnet-5.png")} width="400" />
</div>

:::---
id: config-polygon-on-wallet-link
title: Configure Polygon on Wallet Link
description: Set up Polygon Mainnet on Wallet Link.
keywords:
  - wiki
  - polygon
image: https://wiki.polygon.technology/img/polygon-logo.png
---

import useBaseUrl from '@docusaurus/useBaseUrl';
import Tabs from '@theme/Tabs';
import TabItem from '@theme/TabItem';

:::caution Content disclaimer

Please view the third-party content disclaimer [<ins>here</ins>](https://github.com/maticnetwork/matic-docs/blob/master/CONTENT_DISCLAIMER.md).

:::

---

## Configure Polygon on Walletlink

<Tabs
  defaultValue="mainnet"
  values={[
    { label: 'PoS Mainnet', value: 'mainnet', },
    { label: 'Mumbai Testnet', value: 'mumbai', },
  ]
}>

<TabItem value="mainnet">

1. From the Walletlink home page, select the Settings button

<img src={useBaseUrl("img/walletlink/config-polygon-walletlink-1.jpg")} width="30%" height="30%" />

2. From the setting page, under the Advanced section, select the Active network Option.

<img src={useBaseUrl("img/walletlink/config-polygon-walletlink-2.jpg")} width="30%" height="30%"/>

3. From the Active Networks page, under the ETH section, select the Polygon Mainnet.

<img src={useBaseUrl("img/walletlink/config-polygon-walletlink-3.jpg")} width="30%" height="30%"/>

</TabItem>

<TabItem value="mumbai">

1. From the Walletlink home page, select the Settings button

<img src={useBaseUrl("img/walletlink/config-polygon-walletlink-1.jpg")} width="30%" height="30%" />
```

2. From the setting page, under the Advanced section, select the Active network Option.

<img src={useBaseUrl("img/walletlink/config-polygon-walletlink-2.jpg")} width="30%" height="30%"/>

3. From the Active Networks page, under the ETH section, select the Polygon Mumbai.

<img src={useBaseUrl("img/walletlink/config-polygon-walletlink-4.jpg")} width="30%" height="30%"/>
</TabItem>

</Tabs>---
id: custom-tokens
title: Configure Custom Tokens
description: Configure custom tokens on Metamask.
keywords:
  - wiki
  - polygon
  - custom token
image: https://wiki.polygon.technology/img/polygon-logo.png
---

import useBaseUrl from '@docusaurus/useBaseUrl';

:::caution Content disclaimer

Please view the third-party content disclaimer [<ins>here</ins>](https://github.com/maticnetwork/matic-docs/blob/master/CONTENT_DISCLAIMER.md).

:::

This page demonstrates the process of configuring/adding custom tokens to Metamask.

You can use the same process to add any custom tokens to any network on Metamask. You can refer to [this table](#tokens-and-contract-addresses) to visualize some examples of test tokens with their respective contract addresses.

Adding a custom token to your MetaMask account

First, choose the appropriate network for the new token on your Metamask's home screen. Then click on "Import Tokens".

<img width="300" src={useBaseUrl("img/metamask/develop/add-test-token.png")} />

</br>

It will then navigate you to a new screen. On the Import Tokens screen, copy-paste an address in the Token Address field.

:::info
To illustrate this process, we are using an **ERC20-TESTV4** token on the **Goerli network**. Find other test tokens from other networks [<ins>here</ins>](#tokens-and-contract-adresses).
:::

<img width="300" src={useBaseUrl("img/metamask/develop/token-contract-address.png")} />

The other fields will auto-populate. Click on Add Custom Tokens and then click on Import Tokens. The `TEST` token should now be displayed on your account on Metamask.

<img width="300" src={useBaseUrl("img/metamask/develop/added-token.png")} />

Adding a test ERC1155 token to your MetaMask account

While the Polygon network supports ERC1155, [Metamask does not yet support the standard](https://metamask.zendesk.com/hc/en-us/articles/360058488651-Does-MetaMask-support-ERC-1155-). This update is expected in the fourth quarter of 2021.

Tokens and Contract Addresses

Token	Network	Contract Address
ERC20-TESTV4	Goerli	`0x3f152B63Ec5CA5831061B2DccFb29a874C317502`
MATIC-TST	Mumbai	`0x2d7882beDcbfDDce29Ba99965dd3cdF7fcB10A1e`
ERC721-TESTV4	Goerli	`0xfA08B72137eF907dEB3F202a60EfBc610D2f224b`

| ERC721-TESTV4 | Mumbai | `0x33FC58F12A56280503b04AC7911D1EceEBcE179c` |
---
id: multiple-accounts
title: Create & Import Accounts
description: Create and import multiple accounts.
keywords:
  - wiki
  - polygon
image: https://wiki.polygon.technology/img/polygon-logo.png
---

import useBaseUrl from '@docusaurus/useBaseUrl';

:::caution Content disclaimer

Please view the third-party content disclaimer [<ins>here</ins>](https://github.com/maticnetwork/matic-docs/blob/master/CONTENT_DISCLAIMER.md).

:::

In case you are new to Ethereum and Metamask, here is a guide on how to create multiple accounts and import them:

Create Account

To create multiple accounts, you click on Profile icon on MetaMask and then click on Create Account

<img src={useBaseUrl("img/metamask/create-accounts.png")} />

You can then add an account name and click on Create.

<img src={useBaseUrl("img/metamask/create-new-account.png")} />

You can create more accounts this way. When you create multiple accounts, your addresses will be different at your end.

:::tip Import Account
The importing process on MetaMask is similar to creating a new account. You just have to copy-paste the **Private key** (more information [<ins>here</ins>](https://metamask.zendesk.com/hc/en-us/articles/360015289632-How-to-export-an-account-s-private-key#:~:text=On%20the%20account%20page%2C%20click,click%20%E2%80%9CConfirm%E2%80%9D%20to%20proceed.)), from your local account. Then paste it into the window provided, and select Import.

<img width="300" src={useBaseUrl("img/metamask/develop/import-account.png")} />

:::
---
id: overview
title: Overview of MetaMask
sidebar_label: Overview
description: How you can get started with MetaMask on Polygon
keywords:
  - wiki
  - polygon
  - wallet
  - metamask
image: https://wiki.polygon.technology/img/polygon-logo.png

:::caution Content disclaimer

Please view the third-party content disclaimer [<ins>here</ins>](https://github.com/maticnetwork/matic-docs/blob/master/CONTENT_DISCLAIMER.md).

:::

[MetaMask](https://metamask.io/) is a crypto wallet that can be used in a web browser and on mobile devices to interact with the Ethereum blockchain. It allows you to run Ethereum Dapps (Decentralized Apps) right in your browser without running a full Ethereum node.

Type: Non-custodial/HD
Private Key Storage: User’s local browser storage
Communication to Ethereum Ledger: Infura
Private key encoding: Mnemonic

```
:::warning
 Please Backup your **Secret Recovery Phrase**. If your device breaks, is lost, stolen, or has data corruption, there is no other way to recover it. The Secret Recovery Phrase is the only way to recover your MetaMask accounts. Check more **[<ins>Basic Safety and Security Tips for MetaMask</ins>](https://metamask.zendesk.com/hc/en-us/articles/360015489591-Basic-Safety-and-Security-Tips-for-MetaMask)**.
:::

## Guide to set up MetaMask for Polygon

* [Download & Install MetaMask](/tools/wallets/metamask/tutorial-metamask.md)
* [Configure Polygon on MetaMask](/tools/wallets/metamask/config-polygon-on-metamask.md)
* [Config Custom Tokens](/tools/wallets/metamask/custom-tokens.md)
* [Create & Import Accounts](/tools/wallets/metamask/multiple-accounts.md)

### 1. Set up Web3

#### Step 1

Install the following in your DApp:

  ```javascript
  npm install --save web3
```

Create a new file, name it `web3.js` and insert the following code in it:

```javascript
import Web3 from 'web3';

const getWeb3 = () => new Promise((resolve) => {
  window.addEventListener('load', () => {
    let currentWeb3;

    if (window.ethereum) {
      currentWeb3 = new Web3(window.ethereum);
      try {
        // Request account access if needed
        window.ethereum.enable();
        // Acccounts now exposed
        resolve(currentWeb3);
      } catch (error) {
        // User denied account access...
        alert('Please allow access for the app to work');
      }
    } else if (window.web3) {
      window.web3 = new Web3(web3.currentProvider);
      // Acccounts always exposed
      resolve(currentWeb3);
    } else {
      console.log('Non-Ethereum browser detected. You should consider trying MetaMask!');
    }
  });
});


export default getWeb3;
```

The above file exports a function called `getWeb3()` - the purpose of which is to request MetaMask account's access via detecting a global object (`ethereum` or `web3`) injected by Metamask.

According to [Metamask's API documentation]():

> *MetaMask injects a global API into websites visited by its users at window.ethereum. This API allows websites to request users' Ethereum accounts, read data from blockchains the user is connected to, and suggest that the user sign messages and transactions. The presence of the provider object indicates an Ethereum user.*

In simpler terms, it basically means that having Metamask's extension/add-on installed in your browser, you'd have a global variable defined, called `ethereum` (`web3` for older versions), and using this variable we instantiate our web3 object.

Step 2

Now, in your client code, import the above file:

```javascript
import getWeb3 from '/path/to/web3';
```

and call the function:

```javascript
getWeb3()
  .then((result) => {
```
```

```
      this.web3 = result;// we instantiate our contract next
    });
```

2. Set up account

Now to send transactions (specifically those that alter the state of the blockchain) we'll need an account to sign those transactions. We instantiate our contract instance from the web3 object we created above:

```
this.web3.eth.getAccounts()
.then((accounts) => {
  this.account = accounts[0];
})
```

The `getAccounts()` function returns an array of all the accounts on user's MetaMask, and `accounts[0]` is the one currently selected by the user.

3. Instantiate your contracts

Once we have our `web3` object in place, we'll next instantiate our contracts, assuming you have your contract ABI and address already in place:

```
const myContractInstance = new this.web3.eth.Contract(myContractAbi, myContractAddress)
```

4. Call functions

Now for any function you'd want to call from your contract, we directly interact with our instantiated contract object (which is `myContractInstance` declared in Step 2).

:::tip A quick review

Functions that alter the state of the contract are called `send()` functions. Functions that do not alter the state of the contract are called `call()` functions.

:::

Calling `call()` Functions

```
this.myContractInstance.methods.myMethod(myParams)
.call()
.then (
  // do stuff with returned values
)
```

Calling `send()` Functions

```
this.myContractInstance.methods.myMethod(myParams)
.send({
  from: this.account,gasPrice: 0
})
.then (
  (receipt) => {
    // returns a transaction receipt}
  )
```

---

id: hello title: How to create a MetaMask Wallet? sidebar_label: Hello Metamask description: Learn how to create a MetaMask wallet. keywords: - docs - matic image: https://wiki.polygon.technology/img/polygon-logo.png

import useBaseUrl from '@docusaurus/useBaseUrl';

:::caution Content disclaimer

Please view the third-party content disclaimer here.

:::

If you are wondering how to create a new cryptocurrency wallet, consider creating one by installing the MetaMask extension.

MetaMask is a free and secure browser extension that allows web applications to read and interact with the Ethereum blockchain.

Step 1. Install MetaMask on your browser

To create a new wallet with MetaMask you need to install the extension first. You can install MetaMask for Chrome, Firefox, Brave and Opera browsers.

1. Open https://metamask.io or search for "Metamask extension" using your favorite search engine.

:::note In this tutorial we will be using Google Chrome as an example, but the workflow is the same for all browsers. :::

<img src={useBaseUrl("img/metamask/tools/wallets/metamask-home.png")} />

2. Click **Download** to install MetaMask as a Google Chrome extension.

3. Click **Add to Chrome.**

<img src={useBaseUrl("img/metamask/develop/add-chrome.png")} />

4. Click **Add Extension**.

That's it! You have successfully installed MetaMask extension!

Step 2. Create an account

The next step is to create an account.

1. When the download is finished, you will be able to either **restore a wallet** with a Secret Recovery Phrase or **create a new account**, by creating a new wallet and generating a new Secret Recovery Phrase.

2. You will be asked to create a new password. Create a strong password and click **Create**.

3. MetaMask will then provide some information about the Secret Recovery Phrase and on the next page you will see your phrase.

4. Write the 12-word phrase down on a piece of paper on the same order presented.

:::caution Read carefully Metamask's instructions. Write this phrase on a piece of paper and store in a secure location. If you want even more security, write it down on multiple pieces of paper and store each in 2–3 different locations. You can even memorize this phrase. :::

5. Verify your secret phrase by selecting the previously generated phrase. When done, click **Confirm**.

<img src={useBaseUrl("img/metamask/develop/phrase.gif")} />

By "solving this puzzle" you are confirming that you know your secret phrase.

Congratulations! You have successfully created your MetaMask account. A new Ethereum wallet address was automatically generated for you!--- id: plaid-wallet-onboard title: Plaid Wallet Onboard sidebar_label: Plaid Wallet Onboard description: Wallet Onboard allows you to easily connect your dApp to hundreds of self-custody wallets. keywords: - wiki - polygon - wallet - dapp - integrate - guide image: https://wiki.polygon.technology/img/polygon-logo.png

import useBaseUrl from '@docusaurus/useBaseUrl';

:::caution Content disclaimer

Please view the third-party content disclaimer here.

:::

Plaid Wallet Onboard allows your dApp to connect to hundreds of self-custody wallets, including browser plugin wallets, WalletConnect-compatible mobile wallets, and custom protocols or UX required by hybrid and hardware wallets. It's easier than ever to integrate Metamask, Coinbase Wallet, Rainbow, and more into your dApp.

Wallet Onboard can be configured to automatically ensure that users have their wallets set up to use the Polygon network.

Installation and Prerequisites

Wallet Onboard can be used in any JavaScript-based web dApp, and there is a React library for ReactJS applications as well.

You can read about the entire setup and installation process in the Wallet Onboard docs.

Integrating with Polygon

Wallet Onboard returns an EIP-1193 provider that your dApp can use to request the user's addresses as well as signatures and transactions.

Here's a simple example of the code used to integrate Wallet Onboard on Polygon Mainnet in a vanilla JavaScript dApp:

```
<script src="https://cdn.plaid.com/link/v2/stable/link-initialize.js"></script>

<script>
  const plaidWeb3 = await Plaid.web3();
  const handler = plaidWeb3.createEthereumOnboarding({
```

```
    // retrieve from Plaid Dashboard
    token: 'token-abc',
    chain: {
      rpcUrl: 'https://polygon-rpc.com/',
      chainId: '0x89',
    },
    onSuccess: async (provider) => {
      const accounts = await provider.request({ method: 'eth_accounts' });
    }
  });

  handler.open();
</script>
```

If a user connects their wallet and it is not using Polygon, Wallet Onboard will attempt to add or switch to Polygon automatically, or will instruct the user to switch.

For a complete overview of the configuration options and capabilities of Wallet Onboard, please refer to the [official Wallet Onboard documentation](#).

id: web-wallet-v3-guide title: Polygon Wallet Suite sidebar_label: Usage Guide description: Learn how to use the Polygon Wallet Suite keywords: - wallet - matic - docs - wiki - polygon - guide - web wallet - v3 image: https://wiki.polygon.technology/img/polygon-logo.png

import useBaseUrl from '@docusaurus/useBaseUrl';

A new version of our Polygon Wallet is live. Now called **Polygon Wallet Suite**, our new environment provides a more seamless experience with great improvements on the user interface as well as on its overall experience.

This guide shows instructions to deposit and withdraw funds using Polygon Wallet Suite. For performing those actions, you need to connect a wallet to Polygon's environment. In this tutorial, we used Metamask, but Polygon is integrated with other wallets such as Coinbase, Bitski, Venly and WalletConnect.

:::tip

Please refer to this guide to learn how to connect Polygon to Metamask. :::

Logging into the Polygon Wallet Suite

To log into the Polygon Wallet Suite you need to access the following URL: https://wallet.polygon.technology/.

Once you connect your account with the Web Wallet, you will be taken to the landing page with various means on how to transact with the web wallet. Polygon PoS chain currently offers the following services:

- **Polygon Wallet**: for sending, receiving and storing your assets on the Polygon network
- **Polygon Bridge**: for withdrawals and deposits across networks
- **Polygon Staking**: your go-to place for staking and getting rewards with your MATIC tokens
- and the **Polygon Safe Bridge**.

Click on the Polygon Wallet or Polygon Bridge, and you will see all your token balances on the Polygon Wallet across the bridges.

<img src={useBaseUrl("img/wallet/v3/landing-page.png")} width="100%" height="100%"/>

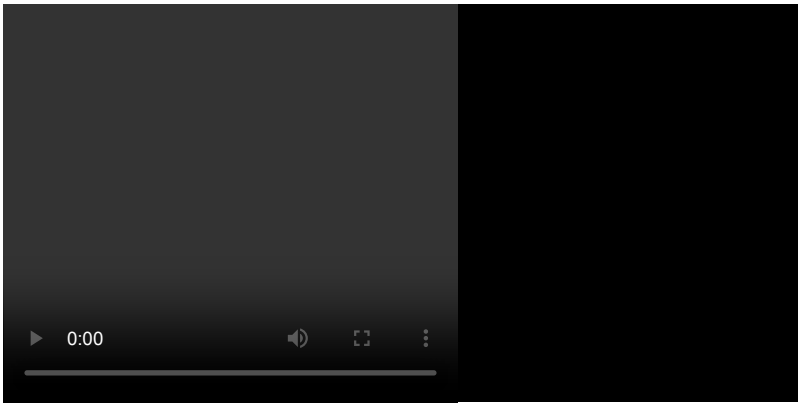
:::tip Metamask

Be attentive to all Metamask's popups. Throughout the deposit and withdrawal processes, you will be prompted with Metamask's popups to confirm transactions, switch networks and other procedures. You can only proceed with those transactions if you confirm the actions on Metamask.

:::

Deposit Funds from Ethereum to Polygon

You can either watch the **video tutorial** here or follow the **step-by-step guide** provided below.



Step-by-Step Guide

1. Click on the **Move Funds from Ethereum to Polygon** button or on the **Deposit** button from any of the token types in the **Token** Balance** section.

<img src={useBaseUrl("img/wallet/v3/deposit/balances.png")} width="1000" height="1000" />

2. You will be redirected to the bridge page where you need to enter the deposit amount.

<img src={useBaseUrl("img/wallet/v3/deposit/bridge.png")} width="100%" height="100%"/>

:::note

The **transfer mode** will be enabled based on the token chosen.

:::

3. Once you have added the amount that you want to deposit, you can then click on the **Transfer** button. After you click on the **Transfer** button, you need to click on the **Continue** button on the next popup.

You will see a **Transfer Overview** popup with an estimate of the total gas required for the transaction:

After that, you can review your transaction details:

4. Once you confirm the transaction, you will see a **Transfer in Progress** popup which will show you the Deposit status. It will take ~7-8 minutes for the tokens to show up on Polygon.

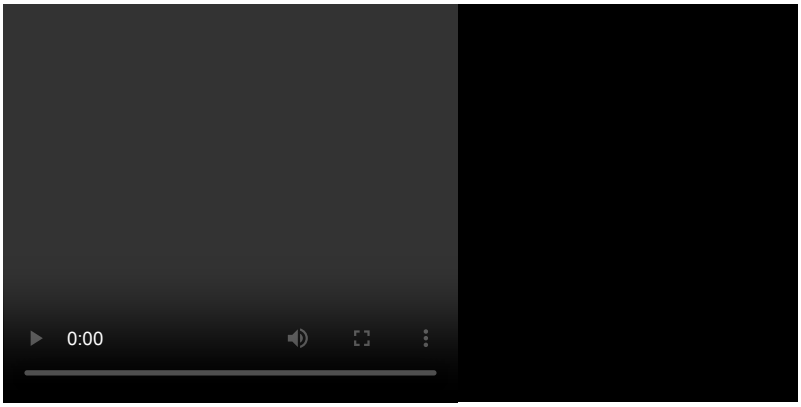
5. After the required time, the transaction will be completed.

You can always check your past and current transactions by clicking on the **Transactions** tab on the left.

Withdraw Funds from Polygon to Ethereum

On Polygon PoS Bridge

Withdrawing funds from Polygon back to the Ethereum Mainnet via PoS Bridge is an extremely simple process. For the funds to be available back on Ethereum it will take from 45 minutes to 3 hours. You can either watch the **video tutorial** here or follow the **step-by-step guide** provided below.



Step-by-Step Guide

1. To withdraw funds, click on the **Withdraw** option on any of the tokens from the **Token Balance** section.

<img src={useBaseUrl("img/wallet/v3/pos/balances.png")} width="100%" height="100%"/>

2. You will be redirected to the Bridge page where you need to enter the Withdrawal amount.

<img src={useBaseUrl("img/wallet/v3/pos/bridge.png")} width="100%" height="100%" />

:::note

The **transfer mode** will be enabled based on the token chosen.

:::

3. Once you have added the amount that you want to withdraw, you can then click on the **Transfer** button. After you click on the **Transfer** button, you need to click on the **Continue** button on the next popup.

You will see a **Transfer Overview** popup with an estimate of the total gas required for the transaction:

After that, you can review your transaction details:

4. Once the transaction is approved, you will see a popup on your screen like this:

The first transaction is to initiate your withdrawal. You need to wait for the checkpoint to arrive. This could take up to 3 hours to complete.

5. Once the checkpoint has arrived, you will need to **confirm the second transaction**. Then, when you have confirmed the second transaction, you will receive your funds back on Ethereum.

Swap for Gas

The MATIC token is used for paying gas fees in the Polygon ecosystem. It's always important to have a minimum amount of MATIC to perform transactions on Polygon. **In fact, 0.01 MATIC can pay for about 20 transactions**. There are a few ways you can acquire MATIC tokens:

- buy them using Fiat On-ramp platforms,
- swap for other tokens, or
- trade them on a supported exchange.

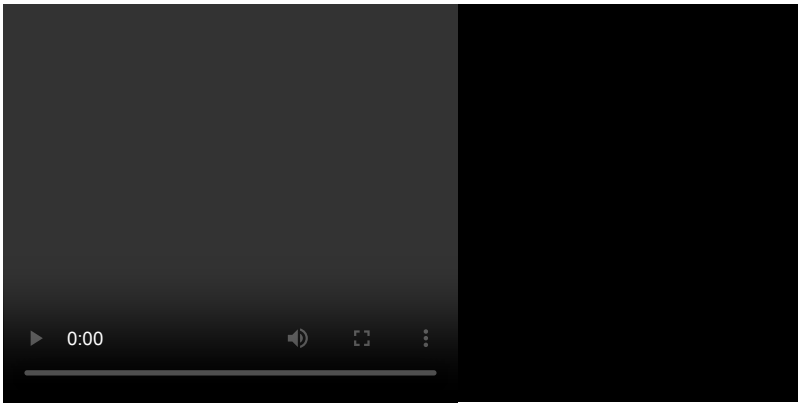
Polygon also offers the possibility of swapping tokens for MATIC. You can currently choose from a range of cryptocurrencies to swap for MATIC, namely ETH, USDC, USDT, and others.

:::tip

This is a gasless transaction. You do not have to pay any gas fees.

:::

You can either watch the video tutorial below or follow the step-by-step guide.



Step-by-step guide

1. Assuming you are on the assets homepage, find the **Swap for Gas** feature on the sidebar on your left.

2. Now, you need to set the amount of MATIC you need, either by choosing one of the given quantities or entering the amount you need (1). You should also select the token that will be swapped for MATIC (2). **The minimum amount for requesting MATIC tokens is 0.5 MATIC**.

3. MetaMask will then prompt you to sign the transaction. If you haven't added the Polygon network to your Metamask, you will be prompted to install it. Please proceed with the installation or check out this guide here.

:::note

If the **Sign** button is not visible, try scrolling down by clicking on the down arrow in the **Message** field.

:::

4. After you signed it, you will be able to transfer the requested amount of MATIC to your wallet.

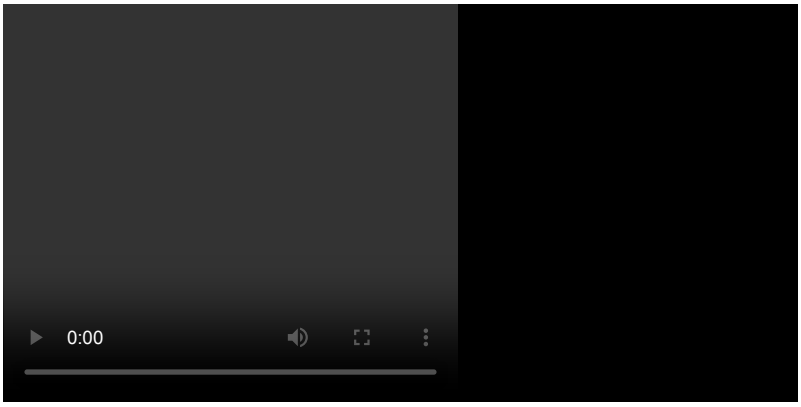
5. A new signature will be requested:

Transaction completed. Remember that you can always verify it on Polygonscan!

Token Swap

You can swap tokens for other tokens using the **Token Swap** feature, which uses decentralized exchanges under the hood. The Token Swap feature may be found in the Wallet Suite side menu.

You can either watch the video tutorial here or follow the step-by-step guide provided below.



Step-by-step guide

1. The token on the top is the one you are swapping. The other one on the bottom is what you are going to receive.

2. Review the transaction before it goes through.

3. **Sign** and confirm the transaction on MetaMask and the transaction will be completed successfully.

Token Lists

You can customize the list of tokens you may see on the Polygon Wallet Suite homepage. For that, click on **Manage Token List**:

There are some default options provided by Polygon, but you can always add new token lists by entering their URLs.

:::tip

You can find some of the token lists here: https://tokenlists.org/

:::

You can also **make your own list**, import it to Polygon Wallet and even share it with the community. To learn more about creating your own token lists, check out this guide to Authoring Token Lists on Github.

:::note

The **Chain ID should be 137** when creating a Polygon list.

:::

If you need to add a token to one of our lists, add a Token Request with the required information, and it'll be reviewed by our team.

:::caution

As it is fairly easy to add custom tokens, users should always do their research and only choose trustworthy projects/tokens.

:::

id: portis title: Portis description: A web-based wallet built keeping easy user-onboarding in mind. keywords: - wiki - polygon - wallet - portis - integrate image: https://wiki.polygon.technology/img/polygon-logo.png

:::caution Content disclaimer

Please view the third-party content disclaimer here.

:::

Portis is a web-based wallet built keeping easy user-onboarding in mind. It comes with a JavaScript SDK that integrates into the dApp and creates a local wallet-less experience for the user. Further, it handles setting up the wallet, transactions, and gas fees.

Like Metamask, it is non-custodial - users control their keys, Portis just stores them securely. But, unlike Metamask, it is integrated into the application and not the browser. Users have their keys associated with their login ID and passwords.

Type: Non-custodial/HD
Private Key Storage: Encrypted and stored on Portis servers
Communication to Ethereum Ledger: Defined by the developer
Private key encoding: Mnemonic

Set up Web3

Install Portis in your dApp:

```
npm install --save @portis/web3
```

Now, register your dApp with Portis to obtain a dApp ID using the Portis Dashboard.

Import `portis` and `web3` objects:

```
import Portis from '@portis/web3';
import Web3 from 'web3';
```

Portis constructor takes the first argument as the dApp ID and the second argument as the network you would like to connect with. This can either be a string or an object.

```
const portis = new Portis('YOUR_DAPP_ID', 'maticTestnet');
const web3 = new Web3(portis.provider);
```

Set up Account

If the installation and instantiation of Web3 was successful, the following should successfully return the connected account:

```
this.web3.eth.getAccounts()
.then((accounts) => {
  this.account = accounts[0];
})
```

Instantiating Contracts

This is how we should instantiate contracts:

```
const myContractInstance = new this.web3.eth.Contract(myContractAbi, myContractAddress)
```

Calling Functions

Calling `call()` Function

```
this.myContractInstance.methods.myMethod(myParams)
.call()
.then (
  // do stuff with returned values
)
```

Calling `send()` Function

```
this.myContractInstance.methods.myMethod(myParams)
.send({
  from: this.account,gasPrice: 0
})
.then ((receipt) => {
  // returns a transaction receipt
})
```---
id: slashauth
title: SlashAuth
description: SlashAuth provides authentication and authorization for dApps.
keywords:
  - wiki
  - polygon
  - slashauth
  - authentication
  - dapp
  - integrate
image: https://wiki.polygon.technology/img/polygon-logo.png

:::caution Content disclaimer

Please view the third-party content disclaimer [<ins>here</ins>](https://github.com/maticnetwork/matic-docs/blob/master/CONTENT_DISCLAIMER.md).

:::

SlashAuth allows developers to quickly authenticate users with their wallets and create **role-based access control based on the on-chain data**. A developer can use SlashAuth to securely log a user in and verify that they have the necessary tokens (ERC20, ERC721, ERC1155) to gain access to the website.

SlashAuth works with existing wallets such as Metamask, Coinbase Wallet, and WalletConnect. It also supports tokens on Ethereum and Polygon with more networks coming soon.

For more documentation, please check out their official [documentation](https://docs.slashauth.com/docs).

Create SlashAuth app

```
Navigate to the [SlashAuth Dashboard](https://app.slashauth.com) and create a new app. Take note of the ClientID and Client
Secret as we will use them below.

## Install the React SDK

Install the Javascript Library via NPM:

```bash
npm i @slashauth/slashauth-react
```

Configure the `SlashAuthProvider` component

The SlashAuth SDK uses [React Context]() to manage state and expose it to components. In order to integrate `/auth` into your app, you must provide the context at the root of your app:

```
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import { SlashAuthProvider } from '@slashauth/slashauth-react';

const root = ReactDOM.createRoot(
  document.getElementById('root') as HTMLElement
);

root.render(
  <SlashAuthProvider
    clientID={/* YOUR CLIENT ID */}
  >
    <App />
  </SlashAuthProvider>
);
```

Add Login with Wallet

The SlashAuth SDK gives you tools to quickly implement authentication via MetaMask wallet in your React application. The simplest implementation is to log the user in directly in their browser. We use the function `loginNoRedirectNoPopup()` from the `useSlashAuth()` hook to accomplish this.

```
import { useSlashAuth } from "@slashauth/slashauth-react";

export const LoginButton = () => {
  const { loginNoRedirectNoPopup } = useSlashAuth();

  return <button className="login-button" onClick={() => loginNoRedirectNoPopup()}>Login With Wallet</button>;
};
```

Authentication information

The SlashAuth SDK exposes information about the current user and their logged in status via data returned by the `useSlashAuth()` hook. Because this data propagates via React Context, any time it changes components will be notified and rerender. Let's create a status component.

```
import { useSlashAuth } from "@slashauth/slashauth-react";
import { useEffect, useState } from "react";

export const LoginStatus = () => {
  const [accessToken, setAccessToken] = useState('');

  const {
    connectedWallet,
    isAuthenticated,
    isLoading,
    getAccessTokenSilently,
  } = useSlashAuth();

  // A way to instantly detect when the access token changes so we
  // can store it in state to display to the screen.
  useEffect(() => {
    if (isAuthenticated) {
      getAccessTokenSilently().then((at: string) => {
        setAccessToken(at);
      });
    } else {
      setAccessToken('');
    }
```
```

```
  }, [getAccessTokenSilently, isAuthenticated]);

  if (isLoading) {
    return (
      <div>
        <span>'Loading...'</span>
      </div>
    );
  }

  return (
    <div>
      <div style={{display: 'block'}}>
        <div>Is Wallet Connected? {connectedWallet ? 'Yes' : 'No'}</div>
        {connectedWallet && <div >Wallet address: {connectedWallet}</div>}
        <div>Is Logged In? {isAuthenticated ? 'Yes' : 'No'}</div>
        {isAuthenticated && <div style={{cursor: 'pointer', color: 'blue', textDecoration: 'underline'}} onClick={() =>
{navigator.clipboard.writeText(accessToken)}}>Click to copy access token</div>}
      </div>
    </div>
  );
}
```

Logout button

The SlashAuth SDK exposes a logout functionality that disconnects the user locally and invalidates their tokens remotely. Let's build a button to add this functionality.

```
import { useSlashAuth } from "@slashauth/slashauth-react";

export const LogoutButton = () => {
  const { logout } = useSlashAuth();

  return <button onClick={() => logout()}>Logout</button>;
};
```

Tying it all together

A simple way to see this all work together is updating your application entry point to display this information.

```
import { useSlashAuth } from '@slashauth/slashauth-react';
import { LoginStatus } from './LoginStatus';
import { LoginButton } from './LoginButton';
import { LogoutButton } from './LogoutButton';


function App() {
  const { isAuthenticated } = useSlashAuth();

  return (
    <div>
      {isAuthenticated ? <LogoutButton /> : <LoginButton />}
      <LoginStatus />
    </div>
  );
}

export default App;
```

Congratulations! You're successfully integrated SlashAuth in your dApp.

You can find the full example of this code in our [GitHub repo](). --- id: torus title: Torus description: Torus is a non-custodial key management system for dApps keywords: - wiki - polygon - torus - wallet - guide - dApp image: https://wiki.polygon.technology/img/polygon-logo.png

:::caution Content disclaimer

Please view the third-party content disclaimer here.

:::

Torus is a user-friendly, secure, and non-custodial key management system for decentralized apps. We're focused on providing mainstream users a gateway to the decentralized ecosystem.

Type: Non-custodial / HD
Private Key Storage: User's local browser storage / Encrypted and stored on Torus servers
Communication to Ethereum Ledger: Infura
Private key encoding: Mnemonic / Social-Auth-login

Depending on your application needs, Torus can be integrated via the Torus Wallet, or by interacting directly with the Torus Network via CustomAuth. For more information, visit the Torus documentation.

Torus Wallet Integration

If your application is already compatible with MetaMask or any other Web3 providers, integrating the Torus Wallet would give you a provider to wrap the same Web3 interface. You can install via a npm package. For more ways and in-depth information, please visit the official Torus documentation on wallet integration.

Installation

```
npm i --save @toruslabs/torus-embed
```

Example

```
import Torus from "@toruslabs/torus-embed";
import Web3 from "web3";

const torus = new Torus({
  buttonPosition: "top-left" // default: bottom-left
});

await torus.init({
  buildEnv: "production", // default: production
  enableLogging: true, // default: false
  network: {
    host: "mumbai", // default: mainnet
    chainId: 80001, // default: 1
    networkName: "Mumbai Test Network" // default: Main Ethereum Network
  },
  showTorusButton: false // default: true
});

await torus.login(); // await torus.ethereum.enable()
const web3 = new Web3(torus.provider);
```

CustomAuth Integration

If you are looking to control your own UX, from login to every interaction, then you can use CustomAuth. You can integrate via one of their SDKs depending on the platform(s) you are building on. For more info, please visit Torus CustomAuth integration.

id: create-wallet title: Create A Wallet on Venly sidebar_label: Create Wallet description: Quick guide on creating Polygon wallets in Venly, previously Arkane Network keywords: - wiki - polygon - create - wallet - arkane network - venly image: https://wiki.polygon.technology/img/polygon-logo.png

:::caution Content disclaimer

Please view the third-party content disclaimer here.

:::

If you are looking for a user-friendly wallet on Polygon network, consider creating a Venly (prev. Arkane Network) wallet. It allows you to enable a recovery mechanism and comes with end-user support via their in-app chat.

Sign Up to Venly

Step 1 → Navigate to the official Venly website and click on the **My Wallet** button in the top right corner.

Sign up to Venly

Step 2 → Click **Create Account** if you are new to Venly. You can also sign up to Venly with your social credentials or with an email-id and password.

Create an account

Step 3 → After signing up with one of your social accounts or with your email-id and password, you will need to accept their terms and conditions.

Step 4 → To secure the wallet, you need to configure a PIN. Keep in mind that your PIN should be between 4 to 6 digits. After entering the PIN, press **Set** and it will take you to the Venly Wallet page.

Create Your Wallet

Now that you've configured your PIN, you are all set to create a Polygon wallet. Select Polygon from the list of various blockchain networks listed on Venly.

Select the Polygon blockchain

To create a new wallet, select **Create New Wallet**. In case if you already have one, select the **Import Wallet** option to retrieve your wallet.

Create a new wallet

Once you've pressed the button, Venly will ask you to confirm the new wallet creation using your PIN (the one you configured a few steps before). After that your wallet will be created and you will be taken to your dashboard.

Wallet Dashboard

Congratulations! You have now created your Polygon wallet using Venly to manage your digital assets on the Polygon network.

---

id: custom-tokens title: Adding Custom Tokens in Venly (prev. Arkane) sidebar_label: Custom Tokens description: Support for custom ERC20/ERC721/ERC1155 tokens on Polygon network keywords: - wiki - polygon - custom tokens - nfts - venly - wallet image: https://wiki.polygon.technology/img/polygon-logo.png

:::caution Content disclaimer

Please view the third-party content disclaimer here.

:::

Fungible Tokens

A developer can easily add support for their custom ERC20 token by creating a small pull request containing the token details toward the Arkane Git repository. Here is an example snippet of the information you must provide:

```
{"name":"SAND","symbol":"SAND","address":"0x3845badade8e6dff049820680d1f14bd3903a5d0","decimals":18,"type":"ERC20"}
```

You can always contact the Venly team via their in-app chat and ask them to add your custom fungible token.

Non-Fungible Tokens (NFTs)

Venly has developed its service in such a way that it will automatically pick up custom created Non-Fungible Tokens (NFTs) if they follow the ERC721 and ERC1155 standard.

The Hulk ERC1155 NFT on Polygon

id: intro title: Introduction to Venly (prev. Arkane) sidebar_label: Introduction description: Integrate an application with Polygon using Venly Wallet keywords: - docs - matic - polygon - venly - connect - blockchain - wallet image: https://wiki.polygon.technology/img/polygon-logo.png

:::caution Content disclaimer

Please view the third-party content disclaimer here.

:::

Venly (prev. Arkane Network) allows you to easily integrate your app to the Polygon blockchain, whether you already have an app integrated with Web3 or are building a new app from scratch. Venly provides a smooth and delightful experience for you and your users on both web & mobile.

Venly will help you in interacting with the Polygon network, create blockchain wallets, create different asset types such as fungible (ERC20), and non-fungible tokens (ERC721 and ERC1155) and interact with smart contracts. Next to a superior developer experience, you can give your users a user-friendly interface.

Each application is unique and has different needs, therefore they provide different ways of interacting with Venly. Applications that have support for Web3 are recommendated to integrate the Venly Web3 provider, others are suggested to use the Venly Widget.

Key Features

- Support Web and Mobile
- Offers social logins
- Offers fiat-on-ramp
- Supports both Polygon and Ethereum

- Supports NFTs (ERC721 and ERC1155) on Polygon
- Easy to integrate using Web3
- Built for a mainstream audience
- Offers in-app customer support

Getting Started

If you already support Web3 technology, you can improve the UX within your application by integrating the **Venly Web3 Provider**, a smart wrapper around the existing Web3 Ethereum JavaScript API.

By making use of our Web3 provider you are able to leverage the full potential of Venly with minimal effort and you will be able to onboard users that are less tech savvy without making them leave your application or download third party plugins. Integrating just takes 2 steps and ~5 minutes.

:::tip

Venly provides a Venly Widget to help developers generate Web 3 support to their applications.

:::

Add the library to your project

Install the library by downloading it to your project via NPM.

```
npm i @venly/web3-provider
```

Add the script to the head of your page.

```
<script src="/node_modules/@venly/web3-provider/dist/web3-provider.js"></script>
```

After adding the javascript file to your page, a global **Venly** object is added to your window. This object is the gateway for creating the Web3 wrapper and fully integrates the widget - Venly Connect.

Initialize the Web3 provider

Add the following lines of code to your project, it will load the Venly web3 provider.

```
Venly.createProviderEngine({clientId: 'Testaccount'}).then(provider => {
    web3 = new Web3(provider);
});
```

The web3 instance now works as if it was injected by Parity or Metamask. You can fetch wallets, sign transactions, and messages.

Congratulations! Your dApp now supports Venly.

:::info To connect with a personalised Client ID, and access our production environment, please request access one using this form.

:::

More about Venly

What makes Venly (prev. Arkane) stand out is their commitment to supporting the ecosystem. Not only their wallet users, by explaining what gas is, or by helping them import an Ethereum wallet into Polygon, but also the developers that are using Venly to build new upcoming products.

Even in their test environments, they add an in-app chat so that developers can directly communicate with the team behind the Venly platform. In case you want to learn more, check out their detailed product documentation.

---

id: network title: Network sidebar_label: Network description: How to switch between Polygon testnet and mainnet. keywords: - wiki - polygon - venly - wallet - switch network - testnet - mainnet image: https://wiki.polygon.technology/img/polygon-logo.png

:::caution Content disclaimer

Please view the third-party content disclaimer here.

:::

Depending on which Arkane environment your application connect, your transactions will be directed to a specific blockchain network. Connect to the Venly (prev. Arkane) production environment and you will be contacting the Polygon Mainnet, whereas connecting to the Venly staging environment will connect you to the Polygon Mumbai testnet.

Below you will find an overview of the public Venly environments and networks to which they connect.

:::note

If you are a developer, it is possible to use different networks or even your own infrastructure. For more information, refer to the Venly documentation.

:::

Environment	Polygon Network	Venly Service Link
Production	Polygon Mainnet	https://app.arkane.network

Staging	Mumbai Testnet	https://staging.arkane.network/

---

id: walletconnect title: WalletConnect description: An open protocol that creates a DApp-Wallet communication. keywords: - wiki - polygon - dapp - wallet - integrate - guide image: https://wiki.polygon.technology/img/polygon-logo.png

:::caution Content disclaimer

Please view the third-party content disclaimer here.

:::

WalletConnect is an open protocol - not a wallet - built to create a communication link between dApps and wallets. A wallet and an application supporting this protocol will enable a secure link through a shared key between any two peers. A connection is initiated by the dApp displaying a QR code with a standard WalletConnect URI and the connection is established when the wallet application approves the connection request. Further requests regarding funds transfer are confirmed on the wallet application itself.

Set Up Web3

To set up your dApp to connect with a user's Polygon Wallet, you can use WalletConnect's provider to directly connect to Polygon. Install the following in your dApp:

```
npm install --save @maticnetwork/walletconnect-provider
```

Install `matic.js` for Polygon integration:

```
$ npm install @maticnetwork/maticjs
```

And add the following code in your dApp;

```
import WalletConnectProvider from "@maticnetwork/walletconnect-provider"

import Web3 from "web3"
import Matic from "maticjs"
```

Next, set up Polygon and Ropsten provider via WalletConnect's object:

```
const maticProvider = new WalletConnectProvider(
  {
    host: `https://rpc-mumbai.matic.today`,
    callbacks: {
      onConnect: console.log('connected'),
      onDisconnect: console.log('disconnected!')
    }
  }
)

const ropstenProvider = new WalletConnectProvider({
  host: `https://ropsten.infura.io/v3/70645f042c3a409599c60f96f6dd9fbc`,
  callbacks: {
    onConnect: console.log('connected'),
    onDisconnect: console.log('disconnected')
  }
})
```

We created the above two provider objects to instantiate our Web3 object with:

```
const maticWeb3 = new Web3(maticProvider)
const ropstenWeb3 = new Web3(ropstenProvider)
```

Instantiating Contracts

Once we have our **web3 object**, the instantiating of contracts involves the same steps as for Metamask. Make sure you have your **contract ABI** and **address** already in place.

```
const myContractInstance = new this.maticWeb3.eth.Contract(myContractAbi, myContractAddress)
```

Calling Functions

:::info

The private key will remain in the user's wallet and the **app does not access it in any way**.

:::

We have two types of functions in Ethereum, depending upon the interaction with the blockchain. We `call()` when we read data and `send()` when we write data.

Calling `call()` Functions

Reading data doesn't require a signature, therefore the code should be like this:

```
this.myContractInstance.methods
  .myMethod(myParams)
  .call()
  .then (
  // do stuff with returned values
  )
```

Calling `send()` Functions

Since writing to the blockchain requires a signature, we prompt the user on their wallet (that supports WalletConnect) to sign the transaction.

This involves three steps:

1. Constructing a transaction
2. Getting a signature on the transaction
3. Sending signed transaction

```
const tx = {
  from: this.account,
  to: myContractAddress,
  gas: 800000,
  data: this.myContractInstance.methods.myMethod(myParams).encodeABI(),
}
```

The above code creates a transaction object which is then sent to user's wallet for signature:

```
maticWeb3.eth.signTransaction(tx)
  .then((result) =>{
    maticWeb3.eth.sendSignedTransaction(result)
    .then((receipt) =>
    console.log (receipt)
  )
})
```

`signTransaction()` function prompts the user for their signature and `sendSignedTransaction()` sends the signed transaction (returns a transaction receipt on success).